IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 7 September 2024, accepted 5 October 2024, date of publication 14 October 2024, date of current version 25 October 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3479237

TOPICAL REVIEW

On the Code Vulnerability Detection Based on Deep Learning: A Comparative Study

GUIPING LI AND YEGE YANG
School of Computer Science and Engineering, Xi'an Technological University, Xi'an 710021, China
Corresponding author: Guiping Li (15693685@qq.com)

This work was supported in part by the Science and Technology Plan Project of Xi'an Beilin District under Grant GX2214, and in part by the Plan Project of Xi'an Science and Technology under Grant 22GXFW0047.

ABSTRACT Deep learning is one of the important methods to detect and fix vulnerabilities in software programs. How to represent code information and how to use artificial intelligence methods to learn and understand code semantics and other information are crucial points in this method. Vulnerability mining analysis based on source code is usually combined with compiler-related techniques to abstract program representations through lexical, syntactic, and semantic analyses, and further combined with data flow analysis, control flow analysis, static symbolic execution, and other techniques to verify the existence of vulnerabilities and identify the location of code defects. To compare the abilities of vulnerability detection methods, we first categorize vulnerability detection methods into two main types based on different intermediate representations: sequence-based and graph-based methods. And then, we further divide sequence-based methods into four categories and distinguish graph-based methods based on whether they employ slicing techniques. Following, through the analysis of specific examples, we compare the advantages and disadvantages of these two methods, and explore the differences and similarities in the neural networks they use. Lastly, we conduct a comparative analysis of the datasets used in the mentioned methods, highlight some challenges in this field, and present our thoughts on potential research directions.

INDEX TERMS Software security, deep learning, program analysis, automated static analysis, code vulnerability detection, vulnerability mining.

I. INTRODUCTION

Software vulnerabilities can cause a network attack and data leakage. Despite lots of efforts have been done to pursue secure programming, vulnerabilities remain prevalent as software becomes more complex and the internet expands.

According to the vulnerability data released by the China National Vulnerability Database of Information Security (CNNVD) in 2022, 24801 new vulnerabilities emerged in 2022 with an increase of 19.28% compared to 2021. However, the proportion of ultra-high risk vulnerabilities is still continuously increasing [1]. Vulnerabilities in software, once exploited by malicious attackers, may cause serious consequences, such as system paralysis and personal privacy data leakage, posing a ransomware risk to the company or posing a threat to public security. Therefore, ensuring the reliability of software has become a current focus of attention to protect internet users from being attacked.

In recent years, Machine Learning(ML) has made significant progress, including the development of Deep Learning(DL) algorithms [2], natural language processing techniques [3], and other data-driven methods that are very effective in detecting software vulnerabilities. ML/DL models excel at discovering subtle patterns and correlations from large datasets, which is crucial in vulnerability detection since vulnerabilities often involve subtle code features and dependencies. The ML/DL model can handle a wide range of data types and formats, including source code [4], [5], [6], [7], [8], [9], [10], [11], textual information [12], and numerical features such as submission features [13]. They can process and analyze these data representations to effectively detect vulnerabilities. This flexibility allows researchers to

The associate editor coordinating the review of this manuscript and approving it for publication was Marco Anisetti.

utilize various data sources and combine different features for comprehensive vulnerability detection.

This paper introduces a class of solutions that use DL techniques to detect vulnerabilities in software programs. ML can be used to identify potential vulnerabilities by training models on vulnerable code samples. These models can then be applied to new software projects, and can automatically identify potentially vulnerable codes based on similar patterns. The use of DL allows for the automatic extraction of features and the implementation of multiple levels of abstraction for representation, thus eliminating the labor-intensive and potentially error-prone task of feature engineering by experts.

In response to the issue of code vulnerability detection, this paper reviews various cutting-edge methods proposed in recent years to analysis and compare the different extraction and representation methods of source code information, as well as the iterative improvement process of the DL models and methods used. Furthermore, we also point out the shortcomings of existing research and emphasize several areas that need attention, thereby suggesting potential avenues for future research.

II. CLASSIFICATION OF CODE VULNERABILITY DETECTION TECHNIQUES

Code vulnerabilities are usually caused by subtle errors, and due to the widespread adoption of open source software and code reuse, which make these vulnerabilities quickly spread. Early detection of vulnerabilities to protect software security has become crucial. Detecting and fixing software vulnerabilities are a complex task because of the wide variety of vulnerabilities and their increasing frequency.

Common code detection techniques can be divided into static analysis techniques, dynamic detection techniques, and a combination of static and dynamic detection techniques. Static analysis techniques can be further subdivided into source code based static analysis techniques and binary program based analysis methods. The source code vulnerability detection methods include intermediate representation based vulnerability detection and logical reasoning based vulnerability detection. There are four kinds of methods for detecting source code based on intermediate representation: code similarity, symbol execution, rule based, and ML based vulnerability detection [14].

In the early days, code similarity was often used to achieve vulnerability detection. Based on code similarity, it was believed that generally similar code had similar vulnerabilities. This core idea was used to detect vulnerabilities caused by code cloning. The drawback of this method is difficult to obtain vulnerability models, and it needs to analyze a lot of error samples manually. Another issue is that it can only detect vulnerabilities with buffer overflow error, making it difficult to cover other types of vulnerabilities. The method based on symbol execution has many limitations, such as path explosion problem, difficulty in solving constraints, and so on.

Rule based or traditional ML approach generally require experts to define rules or features manually to generate vulnerability patterns. Since the above process requires a lot of manual operations, it is inefficient and difficult to accurately describe vulnerabilities, resulting in high rates of false negative and false positive.

DL-based of vulnerability detection methods are currently the forefront of vulnerability detection, which can effectively narrow down the scope of code auditing, avoid expert defined features, and achieve the goal of automatic vulnerability detection [15].

The existing DL-based modeling techniques currently fall into two categories: sequence-based models and graph-based models. In the sequence-based modeling approach, code is represented as a sequence of tokens, which are segments of the code. These tokens may include function call, source code, intermediate code, and assembly code. Various methods analyze these token sequences to detect potential vulnerabilities.

The graph-based modeling approach shows the relationships and structure of code like a map, combining different syntax and meaning-based connections. It can predict vulnerabilities in two main ways: by transforming the graph structure into a sequence or modeling the graph structure directly. Compared with modeling by converting graph structures into sequences, directly using graph neural network (GNN) to model graph structures can more fully learn the dependency relationships between nodes, program structure, and semantic information in graph based code representation. This method can provide more accurate feature vector representations for source code vulnerability detection models. In the method of directly modeling graph structures, one approach is to comprehensively represent the syntax and semantic information related to vulnerabilities at different levels of abstraction by integrating multiple intermediate representations of code, without using slicing techniques. Another approach is to use program slicing techniques to remove information unrelated to vulnerabilities from mixed code representations, in order to improve the accuracy and efficiency of vulnerability detection [16].

This article primarily focuses on vulnerability detection using DL techniques within the context of static analysis, and the technical roadmap of code vulnerability detection based on DL is shown in Figure 1.

The function of compilation includes the process of translating source code into executable computer programs. Due to different program representations at different stages of compilation, as shown in Figure 2, according to the organizational form of code representation, DL based program representation methods can be divided into two categories: sequence based program representation methods and graph based program representation methods. The goals of program representation methods vary, and the suitable vulnerability features for extraction also vary.

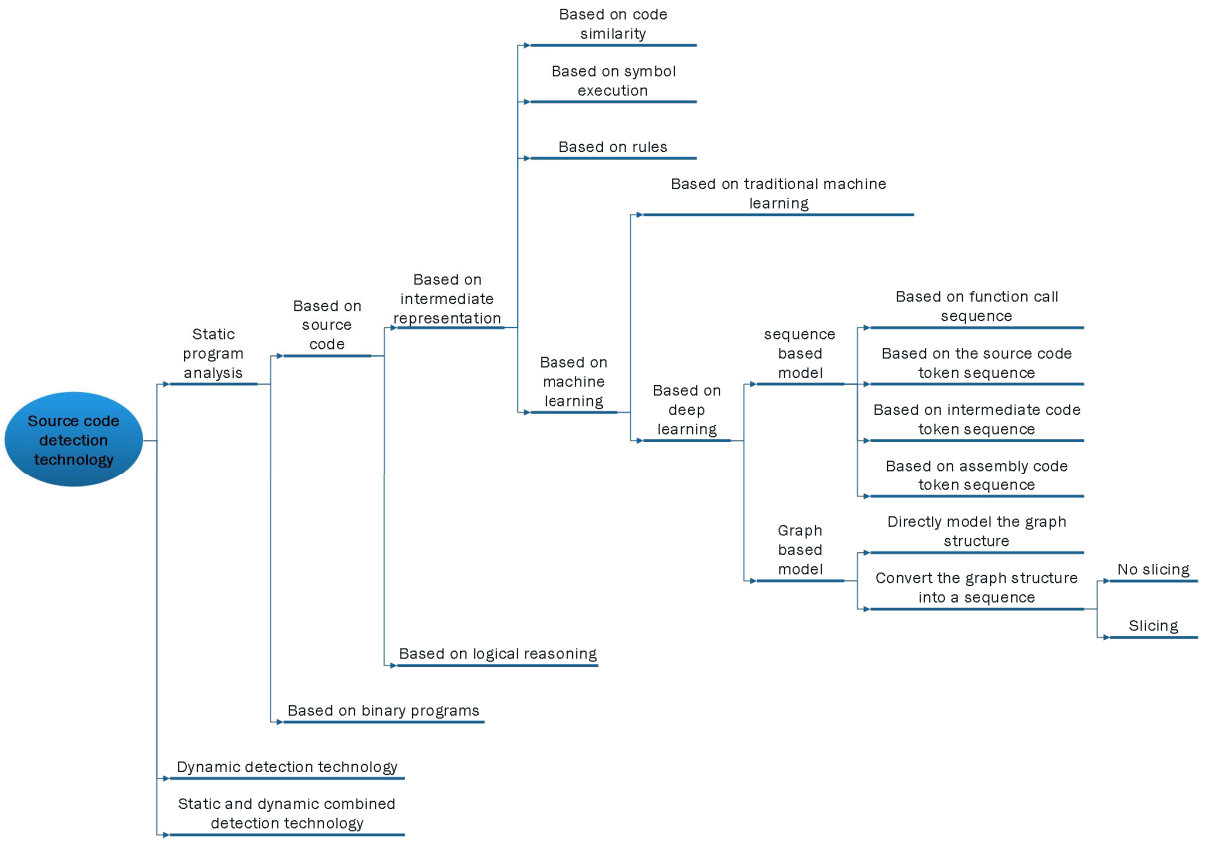

FIGURE 1. Roadmap of code vulnerability detection technology based on DL.

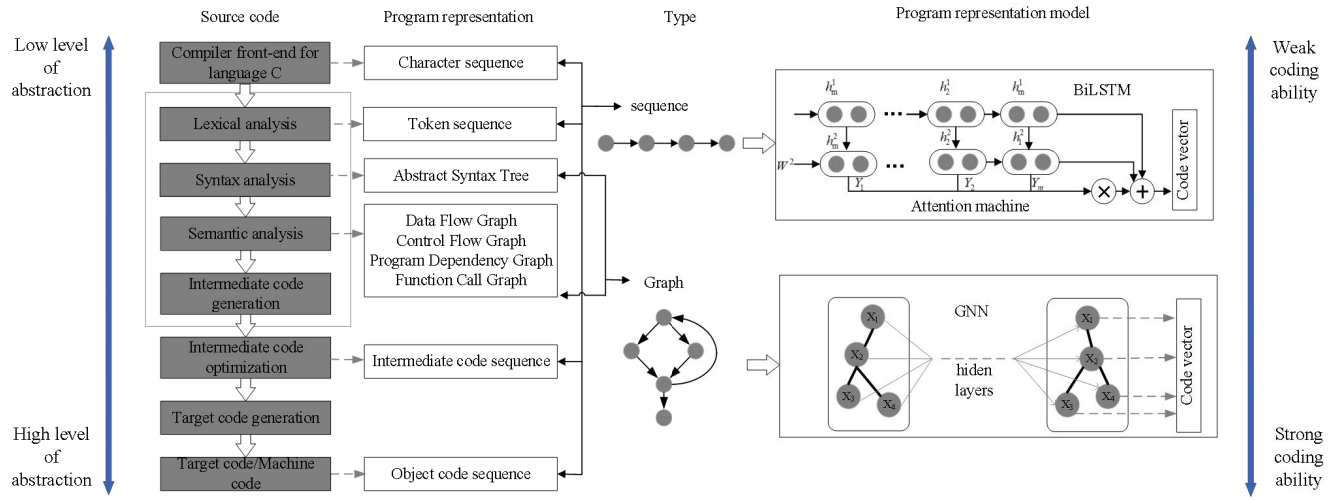

FIGURE 2. Mapping diagram of different code intermediate representations and program representation models.

III. RESEARCH METHODS AND MODELS

A. SEQUENCE BASED MODEL

In the token-based representation, the source code is transformed into a sequence of tokens via compiler-style lexical analysis, and then the sequence is scanned for certain tokens [40]. These models decompose the source code into a sequence of tokens and map each token to an embedded vector space, making it easy to extract the sequential relationships between statements and global semantic information. The process of the

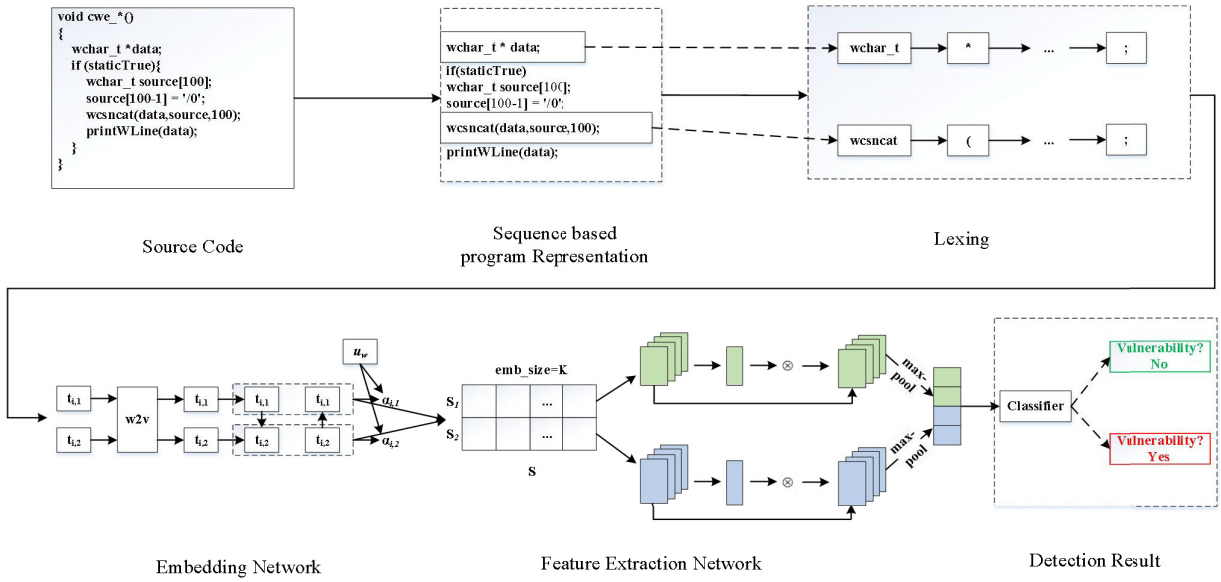

FIGURE 3. The process of the sequence based vulnerability detection method.

sequence based vulnerability detection method is shown in Figure 3.

1) REVIEW OF RELATED WORKS

a: METHOD BASED ON FUNCTION CALL SEQUENCE

The representation method based on function call sequences extracts the function call sequences from the source code by parsing them, and then converts them into vector representations.

To handle function call sequences, Wu et al. [52] used lexical analysis classes commonly used in text processing and mining in Keras. They generated a numerical vector representation of the code by establishing a unique integer index for each word. They used convolutional neural network (CNN), Long Short Term Memory (LSTM), and Convolutional Neural Network-Long Short-Term Memory (CNN-LSTM), and compared them with Multilayer Perceptron (MLP). However, for long sequences, a fixed length of 25 is too small to be fully abstracted for program execution. Word2vec technology can be used to overcome this problem.

b: METHOD BASED ON SOURCE CODE TOKEN SEQUENCE

The program representation method based on source code token sequences converts the code into token sequences through lexical analysis techniques, and then uses word bags, N-gram, or word2vec [50] to convert each token in the Token sequence of the code text into a vector representation. However, this type of method struggles to capture the high-level syntax and semantic information of the code.

In 2018, Rebecca et al. [18] created a custom C/C++ lexer aimed at capturing the relevant meanings of key tokens while maintaining universal representation and minimizing the total token vocabulary. They standardized the lexical representation of code from different software libraries as much as possible to support transfer learning of the entire dataset. Their lexer was able to reduce C/C++ code to representations using a total vocabulary size of only 156 tokens. All base C/C++ keywords, operators, and separators are included in the vocabulary.They combined the neural feature representations of lexed function source code with a powerful ensemble classifier RF.Then with the help of CNN and Recurrent Neural Network (RNN) for functional level source vulnerability classification, they found that CNN+RF has the best performance. But the classification of labels is simpler and cannot consider all the cases.

Yan et al. [19] constructed a corpus that contains 1328690 tokens and adopt skip-gram as training method, finally obtain a vocabulary of 5500 words and the corresponding vector representations.they proposed a Hierarchical Attention Network for Binary Software Vulnerability Detection (HAN-BSVD). The contextual information was enriched by unifying jump addresses and normalizing instructions, which were then tokenized in the lexing step. These normalized instructions were then transformed into token sequences in the lexing step to meet the input requirements of the instruction embedding network. The instruction embedding network, composed of Bidirectional Gated Recurrent Unit (Bi-GRU) and a word attention module, preserved the contextual information. The feature extraction network used Text-CNN with spatial attention modules to capture local features and highlight critical regions. This approach ensured that crucial areas contributed more significantly to the classification results. On the other hand, it also showed that the mixture of different vulnerability categories was more difficult to learn, which required the model to extract more generalized features.

c: METHOD BASED ON INTERMEDIATE CODE TOKEN SEQUENCE

The most commonly used intermediate code in the field of vulnerability detection is Low-Level Virtual Machine Intermediate Representation(LLVM IR), which adopts the form of Static Single Assignment (SSA), which ensures that each variable is defined and used only once. Therefore, intermediate code-based program representations can more accurately encode the semantic features of the code related to control flow and variable def-use relationships [23], and capture more accurate semantic information than source code-based program representations.

In 2022, Li et al. [24] proposed a fine-grained vulnerability detector, VulDeeLocator, which enhanced the accuracy of locating vulnerabilities by utilizing intermediate code to process semantic information. The process began by extracting specific tokens from the program's source code, guided by a predefined set of vulnerability syntax features. These tokens, which included identifiers, operators, constants, and keywords, were referred to as syntax-based Vulnerability Candidates (sSyVC) and represented potential vulnerability elements. Next, the corresponding intermediate code was utilized to integrate statements related to these tokens through data and control dependencies, resulting in a collection known as intermediate code-based Semantics-based Vulnerability Candidates (iSeVC). They also proposed a new dataset, with each sample consisting of intermediate and source code from LLVM. To generate these intermediate codes, user-defined header files and system header files were required. The embedded iSeVC vectors, along with the corresponding vulnerability location matrices, were input into a variant of the Bidirectional Recurrent Neural Network (BRNN-vdl). BRNN-vdl included three key layers: First, the Multiply layer implemented an attention mechanism by selecting outputs associated with vulnerable lines of code, which aided in precise vulnerability localization. Next, the κ-max pooling layer identified the top κ values from the Multiply layer's output. Finally, the average pooling layer computed the mean of the selected values from the κ-max pooling layer. However, due to VulDeeLocator's need to compile the program source code into intermediate code, it could not be used when the program source code could not be compiled.

d: METHOD BASED ON ASSEMBLY CODE TOKEN SEQUENCE

Some researchers represent code based on Token sequences in assembly language.

Tian et al. [9] proposed an intelligent binary code vulnerability detection system BVDetector based on program slicing. They extracted all statements influenced by library/API function call parameters by analyzing the control flow and data flow of binary programs. These statements were then combined to form program slices, which could span different functions. To accomplish this, the Angr tool [39] was employed, utilizing static analysis and symbolic execution techniques for reverse engineering of binary programs. Finally, the extracted program slices were divided into multiple token sequences and added to the sentence list. They designed and constructed a Binary Gated Recurrent Unit (BGRU) network model. However, the system only addressed vulnerabilities related to library/API function calls in binary programs.

2) DISCUSSION

Based on the aforementioned methods, sequence-based vulnerability detection techniques are primarily employed for binary code analysis. These methods effectively capture the program's execution order and logical flow while minimizing the complexity of the analysis. Sequence models demonstrate strong adaptability in processing binary code, particularly in scenarios where source code is unavailable, making them a powerful tool for vulnerability detection.

For sequence based intermediate representation in programs, CNN or RNN is usually used as the representation model to extract vulnerability features. The characteristic of CNN is that it can capture local semantic features. RNNs suffer from the Vanishing Gradient (VG) problem, which can cause ineffective model training.The VG problem can be addressed with the idea of memory cells into RNNs, including the LSTM cell and the Gated Recurrent Unit (GRU) cell. Compared to CNN, RNN and its variants (GRU, LSTM, etc.) have significant advantages in processing sequence data, excelling at capturing sequential semantic information and long dependency information in code context. Studies have shown that Bi-GRU and Bi-LSTM have better performance. Their structures are composed of a combination of forward and backward networks, and their output at each moment is determined by the outputs of the forward and backward RNNs, which can effectively learn the contextual (forward and backward) dependency information of the code. Therefore, many studies use BRNN for code representation learning.

Table 1 shows some information about the sequence based model method mentioned in this article, including datasets, detection granularity, slicing criterion, etc. not mentioned in the article.

B. GRAPH BASED MODEL

Compared to the ''flattened'' feature sets mined from the surface text of source code, many studies use static code analysis tools to extract ''structured'' feature sets as the basis for ML algorithms to learn code patterns. These feature sets originate from different forms of program representations generated through static analysis. For example, AST is a tree structure representation of source code, which is the first step for code parsers to understand the basic structure of a program and check for syntax errors. By retaining the essential structure of the syntax analysis tree while removing superfluous nodes, AST ensures the integrity of expression priorities and semantics. This structured representation

TABLE 1. Reviewed the research on applying sequence based methods to vulnerability detection.

Type	PAPER	Neural Network	Dataset	Year	Embedding Generative Models	Detection Granularity	Slicing Criterion
Method based on Function Call Sequence	Wu et.al[52]	CNN, LSTM and CNN-LSTM	By analyzing 9872 binary programs generated in the '/src/bin' and '/usr/sbin/' directories on a 32-bit Linux machine (real data)	2017	The tokenizer class in Keras.	function-level	/
Method based on Source Code Token Sequence	Russell et.al[18]	CNN and RF	SATE IV Juliet Test Suite (synthetic data) and Debian Linux + GitHub repositories (real data) were tested independently.	2018	word2vec	function-level	/
	Han-BSVD[19]	HAN, Bi-GRU and Text - CNN	Juliet Test Suite dataset (synthetic data) and the ICLR19 dataset (real data) were tested independently.	2021	word2vec	function-level	/
Method based on Intermediate Code Token Sequence	VulDeeLocator [24]	BRNN-vdl	Collect programs on the NVD (real data) and SARD datasets (synthetic data)	2022	word2vec	statement-level	library/API function calls , array usage , pointer usage , and arithmetic expressions in C source programs
Method based on Assembly Code Token Sequence	BVDetector[9]	BGRU	Using the SARD dataset and compiling it into binary programs (synthetic data)	2020	word2vec	statement-level, cross-function detection	library/API function calls in C/ C++ programs

facilitates various analyses, such as syntax and semantic checks, making it an efficient choice for both code understanding and compilation processes. Control Flow Graph (CFG), Data Flow Graph (DFG), and Program Dependency Graph (PDG) help to understand and analyze the structure, execution flow, data flow, and relationships between various program elements of code, and can be used to build variable dependencies and understand program structure. Especially the method proposed by Yamaguchi et al. [25] combines AST, CFG, and PDG into a joint representation called Code Property Graph (CPG), providing a more comprehensive graph structure to describe the attributes of code.This article will introduce the method of directly modeling graph structures. The process of the graph based vulnerability detection method is shown in Figure 4.

1) REVIEW OF RELATED WORKS

a: CONVERT THE GRAPH STRUCTURE INTO A SEQUENCE

The Sequence Graph Hybrid Model utilizes both the graph structure and linear sequence in the program. The information extracted from the graph structure is usually transformed into linear sequences, which can be directly input into machine learning models or deep learning models (such as LSTM, GRU) for analysis and prediction. This transformation makes complex graph structure information easier to be processed by traditional sequence models.

Li et al. [20] proposed VulDeePecker in 2018, which can detect two types of vulnerabilities: buffer errors and resource management errors. They classified library/API function calls based on their input sources into forward and backward calls. Forward calls receive inputs directly from external sources, while backward calls do not. Following this classification, slices of code lines related to these parameters are generated through Data Dependency Graphs(DDG) and assembled into code gadgets based on their semantics. Lexical analysis is then performed to divide the code gadgets in symbolic representation into a series of tokens, including identifiers, keywords, operators, and symbols. These tokens are then converted into vectors using word2vec. To ensure uniform vector length, zeros are padded or vectors are truncated as necessary before being input into a Bi-directional Long Short-Term Memory network (Bi-LSTM). However, VulDeePecker only supported data flow analysis and does not support control flow analysis. Additionally, it could only determine whether a piece of code contains a vulnerability, but it could not determine the type of vulnerability.

Zou et al. [21] proposed the first DL based multi class vulnerability detection system, μVulDeePecker. It could not only determine whether a piece of code has vulnerabilities, but also determine the type of vulnerabilities. Building on VulDeePecker, μVulDeePecker captured global semantic information through improved code gadgets by utilizing System Dependency Graph (SDG) derived from PDG. It identified control and data dependencies within statements by considering both forward and backward slicing. By performing lexical analysis on normalized code gadgets, inferring statement properties from the SDG, analyzing contextual structures, and matching rules in the rule set, code attention is generated. Code attention captures local information and enforces data source checks, critical path checks (such as conditional statements),

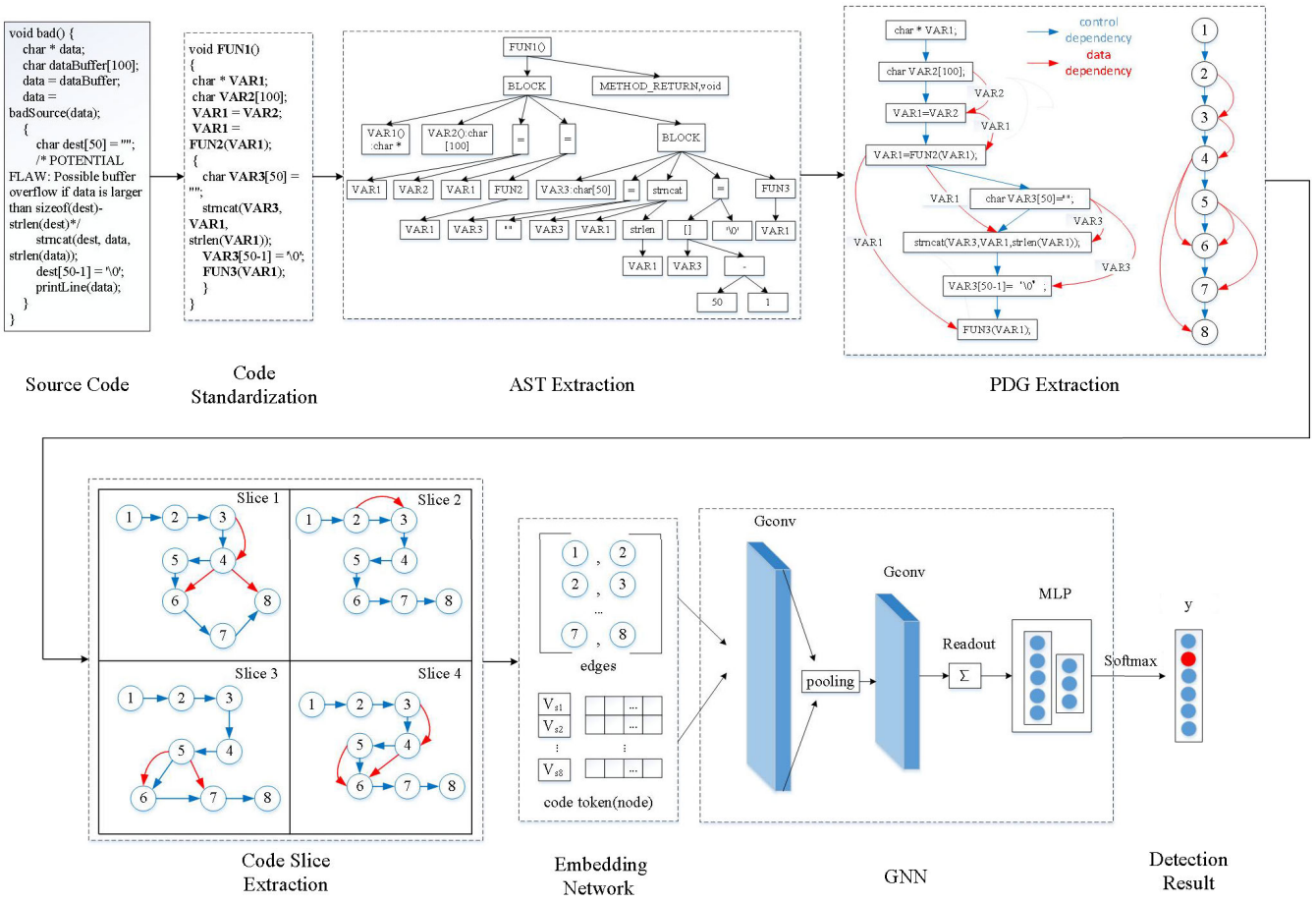

FIGURE 4. The process of the graph based vulnerability detection method.

and dangerous function usage checks, all of which are closely related to vulnerability types. In addition, they used a new network structure called building block Bi-LSTM network, aiming to fuse different types of features from code gadgets and code attention. For the first time, it is proposed to use fusion thinking in vulnerability detection to fuse different types of features and code attention in the code section to adapt to different types of information, without the need to train models separately for each type of vulnerability. However, μVulDeePecker can detect the type of vulnerability, but cannot accurately locate the specific statement of the vulnerability, and can only determine its location in the code fragment.

In 2022, Li et al. [22] introduced the SySeVR framework, which addressed the limitations of the VulDeepecker method. The framework introduced the concepts of syntax vulnerability candidate (SyVCs) and semantic vulnerability candidate (SeVCs). SyVCs were used for preliminary vulnerability detection by representing the program structure with Abstract Syntax Trees (AST). They analyzed statements and tokens within the program to identify and extract code elements that matched specific syntax features, thereby generating potential vulnerability locations such as function calls, array accesses, and pointer operations. To enhance precision in vulnerability detection, SeVCs were generated by leveraging PDG. For each SyVC, forward slices, backward slices, and interprocedural forward and backward slices were created based on these dependencies. These slices were then merged to form SeVCs containing comprehensive semantic information. By integrating semantic information, SeVCs could more accurately pinpoint the true locations of vulnerabilities. Representing the program as a vector suitable for vulnerability detection using Bi-LSTM and Bi-GRU neural networks requires training models separately for each type of vulnerability. As many variables in the program are defined and assigned repeatedly, the same variable may have completely different semantics. The second issue is that this method fails to capture the semantic connections between statements in different files.

The Sequence Graph Hybrid Model performs well in program analysis and vulnerability detection by combining the advantages of graph structures and sequence models. However, its high computational complexity, implementation difficulty, and the possibility of missing key features in complex programs are the main challenges faced by this model.

b: DIRECT MODEL OF THE GRAPH STRUCTURE

i) NOT USING PROGRAM SLICING TECHNOLOGY

Zhou et al. [26] introduced a vulnerability detection model, Devign, in 2019. This model encoded source code functions into CPG, integrating various syntactic and semantic representations such as AST, CFG, and DFG. Additionally, the model considered the natural order of the source code by introducing Natural Code Sequence (NCS) edges within the AST, which connected adjacent code tokens to preserve the code's sequential logic. They utilized the Gated Graph Recurrent Network (GGRN) model, integrating a Conv module for graph-level classification. The Conv module hierarchically learned from node features to capture higher-level representations, thereby enhancing the performance of graph-level classification tasks. However, because Devign operated at the function level and considered only 500 graph nodes within the joint graph structure, its vulnerability detection capabilities were constrained when handling very large functions, leading to notable inaccuracies.

Wang et al. [27] proposed a code structure modeling method called FUNDED in 2021. They first combined probability learning and statistical evaluation to automatically collect high-quality training samples from open source projects. For instance, to prevent existing methods from incorrectly labeling submissions as vulnerability-related due to the presence of keywords like ''check'' and ''NULL'' in the submission information, they employed confidence prediction (CP) to quantify the reliability of the model's predictions. Only predictions with high confidence were accepted, thereby improving the quality of the data. Secondly, they combined the information extracted from AST with Program Control and Dependency Graph (PCDG) to construct a program diagram that can learn and aggregate nine code relationships. Then they extended the Gated Graph Neural Network (GGNN) to model these multiple code relationships extracted from the source code. GGNN stacked four models based on GRU to integrate higher-order neighborhood information across relation graphs. However, the tokens embedded in the statement are simply averaged or fed into a linear layer to obtain the initial node representation, which cannot consider the dependency relationships between tokens in the statement nodes.

In 2021, Li et al. [28] proposed IVDetect. This model integrated information from five dimensions into a feature vector: sub-token sequence, AST subtree, variable names and types, data dependency context, and control dependency context. Graph Convolution Network (GCN) [35] with feature-attention (FA) (referred to as FA-GCN) was then applied to convolve these features to detect vulnerabilities in the code. FA-GCN effectively handled graphs with sparse features and potential noise in the PDG. In addition, this article also used the interpretable method GNNExplainer [47] to explain the classification results of FA-GCN, in order to identify which statements have problems. The core idea was that if removing an edge from the edge set EM affected the model's decision, that edge was crucial for interpreting the detection results and had to be retained. However, IVDetect relied on the processing of PDG to capture code context, but its subgraph-based interpretation model may not have clearly identified the root causes of vulnerabilities, potentially overlooking details.

In order to address the issues of imbalanced and repetitive data in existing datasets, as well as unreasonable model selection, Chakraborty et al. [29] proposed a vulnerability detection method REVEAL and proposed a new dataset. In the feature extraction stage, it first extracted the CPG of the code, and generated an embedded vector representation by combining node types and code snippets. Next, it used GGNN to extract the structural features of the graph, for each vertex in CPG, the GGNN employs a GRU to update the vertex's embedding by integrating the embeddings of its neighboring vertices. During the training phase, the SMOTE algorithm [30] was adopted to solve the problem of sample imbalance. The input features were then projected into a latent space using a MLP, and triplet loss [41] was applied to optimize the distance between samples to enhance separability. Finally, classification detection was performed through the Softmax layer. The model in Chakraborty et al. had several limitations, including data duplication caused by slicing, which made it harder for the model to generalize well to data it had never seen before. Additionally, it struggled with handling data imbalance and failed to capture important semantic dependencies, reducing its effectiveness in accurately detecting vulnerabilities in real-world scenarios.

Hin et al. [31] proposed a new DL framework, LineVD. LineVD took a function from the source code as raw input and divided it into independent statements for processing. Each sample was first tokenized using CodeBERT's [42] pre-trained bytepair encoding (BPE) [43] tokenizer, and then the entire function and its individual statements were passed into CodeBERT to obtain function level and statement level code representations, respectively. Then, LineVD extracted features using CodeBERT, generating n+1 embeddings: one for the entire function and n for the individual statements. It employed the PDG to establish dependency relationships between the represented statements, utilized the Graph Attention Network (GAT) to leverage control and data dependencies, and encoded the raw source code using a Transformer-based model. LineVD's limitations include its inability to effectively propagate information across distant nodes in the code graph, leading to a failure in capturing broader code dependencies.

Wu et al. [32] aimed to achieve scalability and accuracy in large-scale source code vulnerability scanning. They implemented an extensible graph-based vulnerability detection system, VulCNN, by effectively converting the source code of functions into images while preserving program details. They considered the in-degree and out-degree as indicators of the importance of the code. They used Joern [25] to generate the PDG, and each line of code within the PDG

was embedded as a fixed-length vector representation using the Sent2vec [44]. Since images typically consisted of three channels (red, green, and blue), three different centrality measures—degree centrality, Katz centrality, and proximity centrality—were chosen to correspond to these channels. These centrality measures evaluated the importance of each line of code within the function from different perspectives. The results of the centrality analysis and vector processing were then combined into a three-channel image, preserving the structure and detail of the PDG. After the images were generated, vulnerabilities were detected by training a CNN model, which processed the input images using various convolutional filters. It also has some drawbacks, such as the extraction time of PDG for functions being too long and the true negative rate (TNR) not being ideal. Direction ambiguity testing can be used to ease this situation.

ii) USING PROGRAM SLICING TECHNOLOGY

In 2021, the VulSPG proposed by Zheng et al. [33] introduced a new code representation method called Sliced Attribute Graph (SPG), which is a joint graph generated by merging DDG, Control Dependency Graph (CDG), and Function Call Dependency Graph (FCDG). They added two new SyVCs as slicing criteria based on the four SyVCs proposed by Li et al. [22], namely API/library function call (FC), array usage (AU), pointer usage (PU), arithmetic expression (AE), function parameter (FP), and function return statement (FR). The slicing algorithm identified and matched these SyVCs, extracted the relevant code snippets, and incorporated control dependencies, data dependencies, and function call dependencies to generate SPG. During the graph node embedding stage, lexical analysis converted the code statements into embedding vectors, which were further refined using a multi-head self-attention mechanism to extract semantic features. Simultaneously, the node type information was embedded through one-hot encoding, resulting in the initial representation vectors of the nodes. In the graph encoding stage, a Relational Graph Convolutional Network (R-GCN) encoded the SPG and its subgraphs, extracting hidden states of nodes layer by layer and concatenating these states to form the final node representations. Finally, a triple attention mechanism was applied, calculating attention scores at the token, node, and subgraph levels. These scores were used to extract key features, which were then fed into a classifier to detect potential vulnerabilities within the program. However, even with the addition of two new SyVCs, some vulnerability codes remain uncovered by the generated SPG.

To address advanced errors caused by inconsistent business logic and poor programming practices, Cheng et al. [34] proposed a novel embedding method based on DL: DeepWukong. It traversed the PDG forward and backward to generate slices or subgraphs of PDG (XFG), which could effectively maintain the data dependency and control dependency information of the program. After symbolization, the code labels of each XFG node were embedded into fixed-length vectors using Doc2Vec. In the DeepWukong architecture, models such as GCN, GAT, and k-dimensional GNN (k-GNN) employed graph convolutional layers to extract node features. These features were derived not only from the individual node but also by aggregating features from neighboring nodes, enabling the capture of both local and global structural information. The extracted features were subsequently compressed via graph pooling layers and aggregated in the graph readout layer. Finally, the features were processed through a MLP for classification and prediction. However, the vulnerability markers used in DeepWukong may not be perfect, as they were manually annotated and only handle vulnerabilities related to function calls and operator syntax calls.

In order to solve the problem of detecting memory related vulnerabilities in software, Cao et al. [36] proposed MVD, a statement-level memory-related vulnerability detection method based on Flow Sensitive Graph Neural Network (FS-GNN) in 2022. They extended the PDG by incorporating the Call Graph (CG) [37], adding additional semantic information, including function call relationships and return values, to enable interprocedural analysis. For program slicing, they employed reverse slicing based on control dependencies and data dependencies, while forward slicing was conducted solely based on data dependencies. This approach was justified because memory operations, such as allocating but not releasing memory, were often associated with forward data dependencies, whereas forward control dependencies could introduce numerous irrelevant statements. They then used Doc2Vec [45] to encode code statements into fixed-length vectors. During the inference stage, these vectors were optimized using the Stochastic Gradient Descent (SGD) [46] to preserve the semantic information of the code statements. The FS-GNN model integrated statement nodes and multiple flow edges through entity-relation composition operations from the knowledge graph, and applied GraphSMOTE [47] to oversample vulnerable nodes, thereby balancing the distribution by generating synthetic nodes. However, its node labeling was manually done, which might result in some samples being mistakenly labeled.

In 2023, Hu et al. [38] proposed a slice level vulnerability detection and interpretation method based on GNN. They adopted the concept of vulnerability slicing proposed by Li et al. [20] as the guiding principle for slicing generation. They first standardized the C/C++ source code, removed comments and irrelevant information, and then extracted the PDG to preserve code structure information. Next, program slices were generated based on specific vulnerability concerns(such as pointers, arrays, sensitive APIs, etc.). For the generated slices, the word2vec model was used to vectorize each node, extract node features and graph structure features, embed the slice features using a GGNN, and learn vulnerability features in the code by aggregating node and neighborhood information. After completing the detection, the improved GNNExplainer interpreter interpreted the detected vulnerability code and identified specific

lines of vulnerability code. This interpreter optimized the node sorting algorithm to make it more suitable for interpreting vulnerability detection results, thereby improving the accuracy of interpretation and reducing time overhead. However, the current work still has certain limitations. In terms of vulnerability detection, not all vulnerabilities are introduced by the four types of vulnerability concerns mentioned in the article, resulting in some samples being unable to successfully extract slices due to the absence of vulnerability concerns, and thus unable to perform vulnerability detection.

Wen et al. [49] proposed AMPLE to address the problem of existing GNN struggling to capture long-range node dependencies and not fully utilizing multiple edge type information when processing code structure graphs.The framework consisted of two main components: graph simplification and enhanced graph representation learning. Graph simplification reduced redundant information in code structure graphs through type-based and variable-based simplifications. These code structure graphs included the AST, CFG, DFG, and NCS. Enhanced graph representation learning was achieved through two modules. First, the Edge-Aware Graph Convolutional Network (EA-GCN) module leveraged a multi-head self-attention mechanism to enhance node representations by integrating heterogeneous edge information, thereby improving the capture of relationships between distant nodes within the code structure graph. Second, the Kernel-Scaled Representation (KSR) module used two convolution kernels of different sizes (large and small) in parallel to capture both distant and neighboring node relationships, thereby facilitating comprehensive global information learning within the graph structure. However, AMPLE's experimental dataset was limited, the graph simplification method was specific to the C/C++ language, and there may have been deviations in the reproducibility of the baseline methods. These factors could have impacted the generalizability of the results and the accuracy of the comparative analysis.

2) DISCUSSION

For graph based intermediate representations in programs, GNN is usually used to directly model them and use graph level classification results to achieve vulnerability detection. GGNN adds GRU units on the basis of GNN, enabling it to learn both structural and sequential information in the graph simultaneously. Wang et al. [27] used an improved R-GGNN model for representation learning. Compared with GGNN, R-GGNN assigns different learnable weights to different types of nodes and edges, making it more effective for heterogeneous graphs. GCN is also commonly used as a baseline model for vulnerability detection [34]. In some references [33], an improved GCN model R-GCN is used as the main model for feature extraction. Li et al. [28] used another improved GCN based model, FA-GCN, as the main model for vulnerability feature extraction. FA-GCN can effectively handle graphs with heterogeneous features (i.e. not all statements have the same attributes) and remove potential noise in PDGs. In order to assist the model in focusing on learning vulnerability related information in the program, GAT introduces an attention mechanism in the graph, which assigns certain weights to the edges of the nodes to achieve attention to vulnerability related syntax and semantic information in program representation learning.

Table 2 shows some information about the graph based model method mentioned in this article, including datasets, detection granularity, slicing criterion, etc. not mentioned in the article.

IV. DATASET AND EXPERIMENTAL RESULT ANALYSIS

A. REAL-WORLD DATASET

Real data is valuable because it reflects real-world environments, making security measures more robust and reliable. However, it has challenges. Real data often contains noise, making it harder to detect patterns and train machine learning models. The limited amount of real data can reduce the variety of training scenarios, leading to overfitting. Additionally, collecting and preparing real data is time-consuming and expensive, requiring significant resources for cleaning and labeling [16].

For instance, Rebecca et al. [18] utilized real C and C++ functions from Debian Linux and GitHub, achieving an F1 score of 0.566 using a deep learning model based on a combination of Bag-of-Words features and Random Forest. HAN-BSVD [19] achieved an F1 score of 75.35% on the ICLR19 dataset. Wu et al. [52] conducted a study on real Linux system datasets, extracting binary programs from these systems. The model demonstrated an accuracy of 0.836, a true positive rate (TPR) of 0.891, and an F1 score of 0.833, illustrating the advantages of real data in enhancing performance in actual environments. Devign [26] utilized datasets from popular C libraries such as the Linux Kernel, QEMU, and FFmpeg, achieving an F1 score of 0.82. IVDetect [28], applying a fine-grained vulnerability detection model on real-world code from GitHub repositories, achieved an F1 score of 0.87. Although this method excels in granular vulnerability detection, challenges with incomplete or irregular repository data may limit its performance. REVEAL [29] used a real-world dataset curated from large-scale projects like the Linux Debian Kernel and Chromium, achieving an F1 score of 0.4125, which reflects the complexity of real-world data and challenges such as data imbalance and complex vulnerability patterns. LineVD [31], employing the Big-Vul dataset extracted from over 300 open-source C/C++ projects, achieved an F1 score of 0.90, demonstrating high precision. Vu1SPG [33] obtained an F1 score of 69.8% on the FFmpeg dataset and 61.2% on the QEMU dataset. AMPLE [49] achieved F1 scores of 66.94% on the FFmpeg and QEMU dataset, 48.48% on the REVEAL dataset, and 32.11% on the Fan et al. dataset. Vuldetexp [38] employed the Big-Vul dataset for slicing-based vulnerability detection, achieving an F1 score of 0.88, indicating its efficiency in detecting complex vulnerabilities, though the dataset's focus on specific projects may limit its broad applicability.

TABLE 2. Reviewed the research on applying graph based methods to vulnerability detection.

Type	PAPER	Neural Network	Dataset	Year	Embedding Generative Models	Detection Granularity	Slicing Criterion
Convert the graph structure into a sequence	VulDeePecker[20]	RNN and Bi-LSTM	The NVD (real data) and SARD project (synthetic data)	2018	word2vec	statement-level	library/API function calls in C/ C++ programs
	μVulDeePecker[21]	Bi-LSTM	The NVD (real data) and SARD project (synthetic data)	2021	word2vec	statement-level	library/API function calls in C/ C++ programs
	SySeVR[22]	Bi-LSTM and Bi-GRU	The NVD (real data) and SARD project (synthetic data)	2022	word2vec	statement-level, cross-function detection	library/API function calls , array usage , pointer usage , and arithmetic expressions in C/ C++ programs
Directly model the graph structure: Not using program slicing techniques	Devign[26]	GGRN	Manually annotated datasets collected from four popular C libraries: Linux Kernel, QEMU, Wireshark, and FFmpeg. (real data)	2019	word2vec	function-level	/
	FUNDED[27]	GGRN	Collect code from GitHub to supplement standard vulnerability data obtained from CVE (real data) and SARD (synthetic data)	2021	word2vec	function-level	/
	IVDetect[28]	FA-GCN	Devign dataset, REVEAL dataset, Fan et al.'s Big-Vul dataset [54] (real data)	2021	GloVe[17], GRU, Tree-LSTM	function-level	/
	REVEAL[29]	GGNN	Constructed a new dataset using Chromium and Debian's repair commit (real data)	2021	word2vec	function-level	/
	LineVD[31]	Transformer,GAT	Fan et al.'s Big-Vul dataset (real data)	2022	CodeBERT , GloVe, Doc2Vec	statement-level	/
	VulCNN[32]	CNN	The NVD (real data) and SARD project (synthetic data)	2022	sent2vec	function-level	/
	VulSPG[33]	R-GCN	Normalized SySeVR dataset (smei synthetic data) and Devign dataset (real data) were tested independently.	2021	word2vec	function-level, cross-function detection	library/API function calls , array usage , pointer usage , arithmetic expressions ,function parameter and function return statement in C/ C++ programs
Directly model the graph structure: Using program slicing technology	DeepWukong[34]	GCN,GAT, and k-GNN	SARD (synthetic data) and two actual open-source projects (Lua and Redis) (real data)	2021	Doc2Vec	function-level, cross-function detection	library/API function calls , pointer usage , and arithmetic expressions in C/ C++ programs
	MVD[36]	FS-GNN	The CVE (real data) and SARD (synthetic data)	2022	Doc2Vec	statement-level, cross-function detection	library/API function calls , pointer usage in C/ C++ programs
	Vuldetexp[38]	GGNN	Fan et al.'s Big-Vul dataset (real data)	2023	word2vec	function-level	library/API function calls , array usage , pointer usage , and arithmetic expressions in C/ C++ programs
	AMPLE[49]	EA-GCN	Devign dataset, REVEAL dataset, Fan et al.'s Big-Vul dataset (real data)	2023	word2vec	function-level	merge nodes of the same type or using the same variable

In conclusion, the use of real-world datasets—ranging from widely-used open-source projects like the Linux Kernel, FFmpeg, and Redis, to extensive vulnerability databases like NVD and Big-Vul—enables more realistic and practical evaluations. Real-world data provides critical insights into the effectiveness of vulnerability detection models, with F1 scores ranging from 0.3211 to 0.90. These datasets enhance the generalization and applicability of models in real software environments, making detection more reliable. However, real-world datasets often come with limitations, including smaller size, lack of diversity, incomplete annotations, and noise, which can negatively impact model performance and make achieving consistent high accuracy more difficult.

B. SYNTHETIC DATASET

Synthetic data boasts numerous advantages such as large sample sizes, diverse types, minimal noise, and low cost, making vulnerability features easier to learn. However, it significantly differs from real project code in complexity and coverage of programming syntax structures, which hampers its ability to accurately reflect the distribution of vulnerabilities in real-world scenarios. For instance, Rebecca et al. [18] used synthetic code from the SATE IV Juliet Test Suite, where their model, enhanced with a CNN-based approach to extract more granular code features, achieved an F1 score of 0.84. HAN-BSVD [19] excelled on the Juliet Test Suite dataset with an F1 score of 97.15%. BVDetector [9] utilizes the SARD dataset, which covers a range of vulnerability types and somewhat mirrors real detection needs. Nevertheless, the inherent synthetic nature of the dataset may not fully replicate the complexities of actual projects. In binary code vulnerability detection, BVDetector achieved an accuracy rate of 88.2%, a false positive rate of 8.4%, and a false negative rate of 10.6%, with an F1 score of 0.89. These results demonstrate the effectiveness of synthetic data in training models. However, the ability of this data to generalize to unknown real codes may be limited.

C. SEMI-SYNTHETIC DATASET

The semi-synthetic dataset is crafted to facilitate specific purposes, by simplifying and modifying real code. This approach combines the extensive training capabilities of synthetic datasets with the validation offered by real-world data, enabling models to generalize better to complex scenarios.

VulDeeLocator [24] demonstrated high accuracy in positioning vulnerabilities using LLVM IR intermediate code on actual software platforms like the Linux kernel and FFmpeg, achieving up to 96% accuracy and an F1 score of 97% on synthetic datasets.Models like VulDeePecker [20] and SySeVR [22] also leveraged synthetic data to capture a wide range of vulnerability patterns while enhancing their performance in complex code scenarios through real data. These models achieved F1 scores of 0.90 and 0.91, respectively. μVulDeePecker [21] focused on detecting vulnerabilities in C/C++ library function calls, demonstrating excellent vulnerability detection capabilities using the SARD and NVD datasets, with particularly strong performance in multi-class vulnerability detection, achieving an F1 score of 94.22%. FUNDED [27], which combined real data from GitHub and CVE with synthetic samples from SARD, excelled in detecting complex code structures, achieving an F1 score of 0.94. Similarly, VulCNN [32] employed a CNN model to process image-based representations of code, reaching an F1 score of 0.94, while demonstrating excellent scalability in large-scale code analysis. Vu1SPG [33] achieved an F1 score of 94.3% at the slice level. DeepWukong [34], using data from open-source projects like Redis and Lua, achieved an F1 score of 0.956 and an accuracy of 96.5%. Meanwhile, MVD [36], focusing on memory-related vulnerabilities, attained an accuracy of 74.1% and an F1 score of 0.652.

Overall, semi-synthetic data models generally achieve higher F1 scores, ranging from 0.652 to 0.97. This balance between the controlled nature of synthetic datasets and the complexities of real-world data significantly enhances performance. By integrating synthetic and real-world elements, these models gain the ability to perform well in practical applications while retaining adaptability and robustness. This combined approach has proven effective in improving the generalization of vulnerability detection models in real-world.

Based on our analysis and results, we find that real-world data reflects the complexity of software projects, leading to lower F1 scores. In contrast, synthetic datasets focus on specific vulnerabilities, offering better control over data quality and distribution, which typically results in higher F1 scores. However, models trained on synthetic data tend to generalize poorly to real-world scenarios. Semi-synthetic datasets, which combine real and synthetic data, perform the best, achieving F1 scores above 0.9 for most models. This approach balances the complexity of real data with the control of synthetic data, improving detection performance. Figure 5 shows the F1 scores for different data sources.

Additionally, in various studies, researchers have employed strategies to address data challenges beyond using program slicing to filter irrelevant nodes and reduce noise. To handle data imbalance, methods such as weighted loss functions, SMOTE, GraphSMOTE oversampling, weight adjustment, and random undersampling have been applied. These approaches either assign higher weights to minority class samples or generate synthetic samples to balance class distribution, improving the model's ability to learn minority features. For duplicate samples, researchers use strict removal techniques such as code similarity detection and eliminating samples with identical lexical representations to prevent overlap between training and testing data, reducing overfitting risks. In summary, addressing these issues enables the model to better learn diverse features and enhances its generalization ability, allowing it to not only perform well on training data but also accurately infer on unseen data. By reducing the risks of overfitting and data leakage, these strategies improve the model's adaptability and robustness in complex, real-world scenarios.

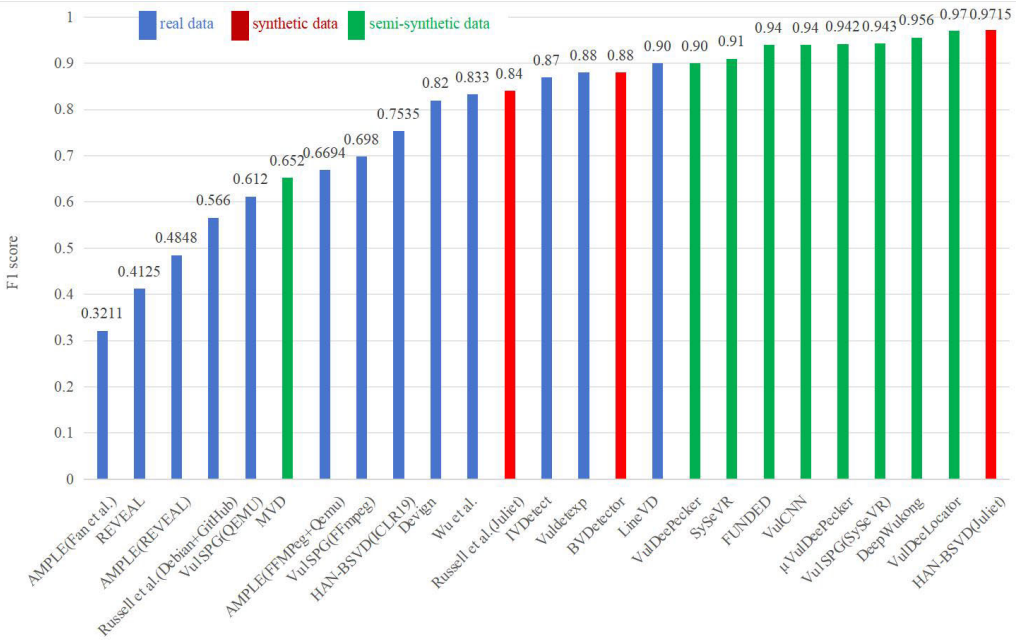

FIGURE 5. Comparison of F1 scores for vulnerability detection methods using real, synthetic, and semi-synthetic data.

V. CONCLUSION

Based on the above description, we can see that sequence-based modeling methods assume a linear order among elements in the input sequence, which is effective for capturing the lexical order and token information of the program. This approach enables sequence models, such as RNNs or LSTMs, to learn patterns within token sequences, making them well-suited for analyzing the lexical flow of the code. However, representing a program as a linear sequence oversimplifies the inherent structural information, overlooking critical aspects like control flow, data dependencies, and hierarchical relationships between code components. Such simplification can lead to the loss of important structural information, which is crucial for accurately identifying vulnerabilities, as the linear representation struggles to capture the multi-dimensional context and interactions present in complex codebases. Therefore, while sequence-based models can efficiently process token sequences, they may face limitations in generalizing to more complex program behaviors. This highlights the need for complementary approaches, such as graph-based models, to fully understand and detect vulnerabilities in code.

The graph-based modeling approach treats code as graphs, integrating syntactic and semantic dependencies to preserve complex information like logical structure and relationships. However, when applied to large-scale programs, these methods face challenges due to the complexity of graphs with densely connected edges, making feature extraction difficult. GNNs struggle with scalability, as large graphs require significant computational resources. Additionally, noisy edges can obscure critical patterns, and capturing long-range dependencies is challenging, limiting the understanding of global code structure. Ensuring generalizability across diverse codebases also remains a challenge, highlighting the need for efficient graph simplification and improved aggregation methods.

In addition, due to hardware limitations, the input vector length of the model usually needs to be fixed, resulting in code with large scale and vulnerability statements distributed at the end of the function being truncated when exceeding the limit, making it impossible for the model to learn the complete semantic information of the code. Meanwhile, code elements in programs, such as tokens, statements, etc., often have a large number of contextual dependencies. The model needs to selectively retain and learn contextual dependencies related to vulnerabilities to effectively identify vulnerability patterns. However, vulnerability patterns in real projects are often complex, and learning based methods are difficult to accurately learn and express the deep vulnerability semantics in the code. Therefore, how to construct a detection model that can extract complex vulnerability features has become a challenge for learning based vulnerability detection methods.

The analysis of existing research reveals that most vulnerability detection methods focus on either coarse-grained detection at the function or slice level, or fine-grained detection at the statement level. We recommend using a multi granularity vulnerability detection method, which first conducts preliminary checks on the code at the function or slice level, filters out parts that may have vulnerabilities, and then conducts fine-grained detection at the statement level for these parts to more accurately identify vulnerabilities. At the same time, the optimization objective for fine-grained vulnerability detection should not be to maximize the probability

of predicting an entire code segment as a true vulnerability label (1 or 0), as in coarse-grained detection. Instead, it should focus on identifying the set of statements that cause vulnerabilities by maximizing the Intersection over Union (IoU) between the actual and predicted sets of vulnerable statement locations. Additionally, leveraging the implicit vulnerability patterns within fine-grained labels to learn the co-occurrence and conditional dependency relationships between vulnerable statements can further enhance the accuracy of fine-grained vulnerability detection in code.

In deep learning based vulnerability detection methods, security domain knowledge and static program analysis techniques also can be utilized to enhance model design

while helping developers understand and fix vulnerabilities. Static program analysis techniques, such as symbolic execution and taint analysis, are capable of capturing control flow and data flow dependencies in code. Meanwhile, security domain knowledge—such as vulnerability patterns for buffer overflows and SQL injection, attack vectors, secure coding practices, and information from known vulnerability databases—provides deep insights and rich features for deep learning models. Together, these elements enable more accurate detection and interpretation of potential security issues.

ACKNOWLEDGMENT

The authors would like to thank the editor and reviewers for their constructive comments.

REFERENCES

[1] (2023). *CNNVD: Annual Report on the Status of Network Security Vulnerabilities*. [Online]. Available: https://www.cnnvd.org.cn/home/globalSearch?keyword=%E5%B9%B4%E5%BA%A6%E7%BD%91%E7%BB%9C%E5%AE%89%E5%85%A8%E6%BC%8F%E6%B4%9E%E6%80%81%E5%8A%BF%E6%8A%A5%E5%91%8A

[2] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2018, pp. 4171–4186.

[4] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," 2017, *arXiv:1708.02368*.

[5] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 67–85, Jan. 2021.

[6] S. Jeon and H. K. Kim, "AutoVAS: An automated vulnerability analysis system with a deep learning approach," *Comput. Secur.*, vol. 106, Jul. 2021, Art. no. 102308.

[7] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, "CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation," *IEEE Trans. Depend. Secure Comput.*, vol. 19, no. 1, pp. 438–451, Jan. 2022.

[8] T. Shippey, D. Bowes, and T. Hall, "Automatically identifying code features for software defect prediction: Using AST N-grams," *Inf. Softw. Technol.*, vol. 106, pp. 142–160, Feb. 2019.

[9] J. Tian, W. Xing, and Z. Li, "BVDetector: A program slice-based binary code vulnerability intelligent detection system," *Inf. Softw. Technol.*, vol. 123, Jul. 2020, Art. no. 106289.

[10] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 297–308.

[11] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 499–510.

[12] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 34–45.

[13] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, Apr. 2019.

[14] Q. Jiao, "Research on source code vulnerability detection technology based on graph computing," M.S. thesis, Univ. Electron. Sci. Technol. China, 2021, doi: 10.27005/d.cnki.gdzku.2021.004494.

[15] X.D. Yan, "Research on software vulnerability detection technology based on static taint analysis and deep learning," M.S. thesis, Harbin Inst. Technol., 2022, doi: 10.27061/d.cnki.ghgdu.2021.003610.

[16] X. Su, W. Zheng, and Y. Jiang, "Research and progress on learning-based source code vulnerability detection," *J. Comput. Sci. Technol.*, vol. 47, no. 2, pp. 337–374, 2024.

[17] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.

[18] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[19] H. Yan, S. Luo, L. Pan, and Y. Zhang, "HAN-BSVD: A hierarchical attention network for binary software vulnerability detection," *Comput. Secur.*, vol. 108, Sep. 2021, Art. no. 102286.

[20] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, *arXiv:1801.01681*.

[21] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "$\mu\mu$VulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Depend. Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, Sep. 2021.

[22] 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 Trans. Depend. Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul. 2022.

[23] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Jan. 2018.

[24] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: A deep learning-based fine-grained vulnerability detector," *IEEE Trans. Depend. Secure Comput.*, vol. 19, no. 4, pp. 2821–2837, Jul. 2022.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604.

[26] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 32, 2019, pp. 10197–10207.

[27] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 1943–1958, 2021.

[28] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2021, pp. 292–303.

[29] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[30] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, Jun. 2002.

[31] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, May 2022, pp. 596–607.

[32] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 2365–2376.

[33] W. Zheng, Y. Jiang, and X. Su, "Vu1SPG: Vulnerability detection based on slice property graph representation learning," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2021, pp. 457–467.

[34] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–33, Jul. 2021.

[35] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
[36] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 1456–1468.
[37] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, pp. 216–226, May 1979.
[38] Y. Hu, S. Wang, and Y. Wu, "A graph neural network-based method for slice-level vulnerability detection and explanation," *J. Softw.*, vol. 34, no. 6, pp. 2543–2561, 2023, doi: 10.13328/j.cnki.jos.006849.
[39] X. Wang, C. Hu, R. Ma, and X. Gao, "A survey of the key technology of binary program vulnerability discovery," *Netinfo Secur.*, vol. 17, pp. 1–13, 2017.
[40] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 201–213.
[41] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," in *Proc. Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, 2019, pp. 478–489.
[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics: EMNLP*, 2020, pp. 1536–1547.
[43] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (Long Papers)*, vol. 1, 2016, pp. 1715–1725.
[44] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-Gram features," 2017, *arXiv:1703.02507*.
[45] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31th Int. Conf. Mach. Learn. (ICML)*, Beijing, China, 2014, pp. 21–26.
[46] N. K. Sinha and M. P. Griscik, "A stochastic approximation method," *IEEE Trans. Syst. Man, Cybern.*, vol. SMC-1, no. 4, pp. 338–344, Oct. 1971.
[47] T. Zhao, X. Zhang, and S. Wang, "GraphSMOTE: Imbalanced node classification on graphs with graph neural networks," in *Proc. 14th ACM Int. Conf. Web Search Data Mining*, Mar. 2021, pp. 833–841.
[48] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "GNNExplainer: Generating explanations for graph neural networks," in *Proc. NIPS*, 2019, pp. 9240–9251.
[49] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2275–2286.
[50] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. IEEE/ACM 17th Int. Conf. Mining Softw. Repositories (MSR)*, May 2020, pp. 508–512.
[51] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Neural Inf. Process. Syst. (NIPS)*, 2013, pp. 3111–3119.
[52] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Dec. 2017, pp. 1298–1302.

GUIPING LI received the B.S. degree from Zhengzhou University, China, the M.S. degree from Xi'an University of Science and Technology, China, and the Ph.D. degree from Xidian University, China. She is currently an Associate Professor with the School of Computer Science and Engineering, Xi'an Technological University. Her current research interests include communication information theory, channel coding theory, and their applications.

YEGE YANG received the B.S. degree in computer science and technology from Xi'an Technological University, China, in 2022, where she is currently pursuing the M.S. degree. Her current research interests include artificial intelligence and code vulnerability detection.