HgtJIT: Just-in-Time Vulnerability Detection Based on Heterogeneous Graph Transformer

Xiaobing Sun, *Member, IEEE*, Mingxuan Zhou, Sicong Cao, Xiaoxue Wu, Lili Bo, Di Wu, *Senior Member, IEEE*, Bin Li, and Yang Xiang, *Fellow, IEEE*

***Abstract*—Vulnerability detection plays a crucial role in the software development lifecycle. Commit-level vulnerability detection aims to detect whether the changed code contributed to potential vulnerabilities by the developer when submitting the code, which is also referred to as Just-In-Time (JIT) vulnerability detection. Previous JIT vulnerability detection approaches relied on code metrics and textual features, which were unable to effectively characterize vulnerability-contributing commits (VCCs). Recently, CodeJIT (a code-centric learning-based approach) has been proposed to detect vulnerability at the commit-level. However, CodeJIT still has its limitations: imprecise feature representation, static code embedding, and underutilized heterogeneous information. In this paper, we propose HgtJIT, a JIT vulnerability detection approach based on a Heterogeneous Graph Transformer (HGT) in order to address several limitations of the state-of-the-art CodeJIT approach. We propose diffPDG to represent code changes and use the CCT5 model (the latest feature encoder pre-trained on a large-scale code change corpus) to embed graph nodes to generate the most meaningful vector representations. In addition, we employ HGT to adequately utilize heterogeneous information of the graph to learn vulnerability features. Extensive experiments have shown that HgtJIT is the best-performing model, with $F1$ and AUC improvement of 14.6%–37.5% and 12.2%–53.7% compared to the baseline model.**

***Index Terms*—Just-in-time vulnerability detection, code change representation, pre-trained model, heterogeneous graph learning.**

I. Introduction

Software vulnerabilities, sometimes called security bugs, can be exploited by hackers to perform malicious behaviors, posing threats to individuals, businesses, and social public security. For instance, the Sunburst vulnerability, which emerged in 2020, allowed hackers to infiltrate the computer systems of hundreds of companies and government institutions by manipulating the updates of SolarWinds software.[1] This incident raised significant concerns because it enabled hackers to access a vast amount of sensitive information, including data from government agencies and businesses. The inestimable consequences highlight the importance of timely and accurate vulnerability detection.

Existing efforts: Traditional approaches leverage static code analyzers [1], [2], [3] to hunt security vulnerabilities hidden in programs. However, even state-of-the-art tools have achieved limited success in realistic scenarios as they heavily rely on hand-crafted vulnerability specifications and rules, which are *time-consuming* and *error-prone*. Benefiting from the tremendous progresses of Deep Learning (DL) in code understanding, recent years have witnessed an increasing in the popularity of learning-based vulnerability detection approaches [4], [5], [6], [7], [8], [9], [10], [11] because they can automatically learn implicit code patterns from vulnerable *files* or *functions* without human intervention. However, it is crucial to identify vulnerabilities as soon as possible because late fixes and consequent damage due to affected systems are severe. Toward that, **Just-In-Time (JIT) vulnerability detection** approaches have been proposed to detect vulnerabilities at the *commit-level*. It is more practical because it can timely catch vulnerabilities once a new code change is committed to the repository, fundamentally reducing the potential security risks. Early works [12], [13] generally take code metrics and textual features as input to train a Machine Learning (ML)-based detection model. Despite their effectiveness, a recent empirical study [14] pointed out that simple statistical or textual features fall short in characterizing vulnerability commits because they often involves changes in source code structure, modifications in program dependencies, and the evolution of code relations. In response to this, an emerging research CodeJIT [15] proposed a code-centric JIT vulnerability detection approach. It constructs a Code Transformation Graph (CTG) based on changed code, related unchanged code, and program dependency relations to represent the semantics of code changes. Subsequently, it employs Graph Neural Networks (GNNs) to learn vulnerability features for classification.

Limitations: While promising, it still has several limitations:

Received 17 August 2024; revised 29 May 2025; accepted 3 July 2025. Date of publication 7 July 2025; date of current version 4 November 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62002309 and Grant 62202414, in part by the Jiangsu "333" Project, and in part by the Open Foundation of Yunnan Key Laboratory of Software Engineering under Grant 2023SE201. *(Corresponding author: Sicong Cao.)*

Xiaobing Sun, Mingxuan Zhou, Sicong Cao, Xiaoxue Wu, and Bin Li are with the School of Information Engineering, Yangzhou University, Yangzhou 225009, China (e-mail: xbsun@yzu.edu.cn; DX120210088@yzu.edu.cn; xiaoxuewu@yzu.edu.cn; lb@yzu.edu.cn; MZ120220958@stu.yzu.edu.cn).

Lili Bo is with the School of Information Engineering, Yangzhou University, Yangzhou 225009, China, and also with Yunnan Key Laboratory of Software Engineering, Kunming 650504, China (e-mail: lilibo@yzu.edu.cn).

Di Wu is with the School of Mathematics, Physics and Computing, University of Southern Queensland, Toowoomba, QLD 4350, Australia (e-mail: di.wu@unisq.edu.au).

Yang Xiang is with the Swinburne University of Technology, Hawthorn, VIC 3122, Australia (e-mail: yxiang@swin.edu.au).

Digital Object Identifier 10.1109/TDSC.2025.3586669

[1]https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor

1545-5971 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

- *Imprecise Feature Representation:* Inspired by the effectiveness of Code Property Graphs (CPG) [16] in modeling vulnerability features, CodeJIT merges the CPGs of the pre- and post-change files, and prunes vulnerability-irrelevant nodes that do not have structure or dependency relations with the changed nodes to construct Code Transformation Graph (CTG). However, since the scale of CPGs grows exponentially as the lines of code increase, blindly considering *all* contextual nodes still introduce numerous noise information. As reported in [17], current GNN-based vulnerability detectors show obviously performance degradation when the CPG of an input function has more than 50 nodes, let alone the CTG which is derived from files.
- *Static Code Embedding:* CodeJIT uses Word2Vec [18] trained on a project-specific dataset to generate vector representations of source code for model training. Such static code embeddings have been proven to be susceptible to the problem of *Out-Of-Vocabulary* (OOV), leading to poor generalization ability [19].
- *Underutilized Heterogeneous Information:* CodeJIT utilizes CTG as input to relational graph neural networks for learning vulnerability features. However, the code change graph not only contains information on edge types (e.g., data dependencies, control dependencies) but also includes information on node types (e.g., addition and deletion of code change). Relational graph neural networks simply distinct non-sharing weights for edge type alone, making them insufficient to capture heterogeneous graph's properties [20], [21], leading to the model being unable to be aware of the semantic change relations brought about by the change information and affecting the learning of vulnerability features.

Our Work: In response to these limitations, we propose a JIT vulnerability detection approach called HgtJIT based on Heterogeneous Graph Transformer (HGT) [20]. First, We construct a novel change-level graph representation named diffPDG by merging the Program Dependency Graphs (PDG) [22] of pre-change and post-change files, and perform program slicing based on changed nodes to focus on vulnerability-relevant parts. Compared to CTG, diffPDG pays more attention on semantics relations between statements, and limits the context scope to alleviate the negative impact of noise nodes. Second, we use the CCT5 model [23], which is pre-trained on the large-scale code change corpus, to embed graphs nodes, effectively alleviates the OOV problem. Third, we employ the HGT model [20] to learn vulnerability features. HGT assigns a unique attention weight for each type of node and edge node, sufficiently utilizing the heterogeneous information of code changes.

Evaluations: We evaluate HgtJIT and five state-of-the-art baselines, including VCCFinder [12], DeepJIT [24], CC2Vec [25], CCT5 [23], and CodeJIT [15], on a real-world dataset of 10,984 commits, in which 31.5% samples are Vulnerability-Contributing Commits (VCC). The experimental results show that HgtJIT achieves optimal performance in JIT vulnerability detection. Compared to state-of-the-art approaches, our approach has shown improvements by 14.6%-37.5% in $F1$ and 12.2%-53.7% in AUC. This indicates that HgtJIT can effectively detect potential VCCs in practice.

Contributions: In summary, the main contributions of this paper are as follows:

- We propose a novel graph-based representation of code changes called diffPDG, which focuses more on contextual nodes semantically highly related to changed nodes.
- We propose a novel approach HgtJIT, which fully preserves change domain knowledge and utilizes heterogeneous information of diffPDG for JIT vulnerability detection.
- We extensively evaluate our proposed approach on a large-scale dataset, demonstrating its effectiveness on JIT vulnerability detection.
- To facilitate further research, we have made our source code, models, and datasets available.[2]

II. Background

A. JIT Software Vulnerability Detection

In an environment where software iterations occur frequently, JIT vulnerability detection becomes crucial. JIT vulnerability detection promptly detects and reports potential vulnerabilities when developers commit code. This approach mitigates the risk of introducing vulnerabilities, it often deals with smaller code segments, enabling developers to swiftly find and fix problems. CodeJIT [15] introduces a state-of-the-art, code-centric JIT vulnerability detection approach, which consists of three steps:

Step 1 (CTG Construction): CodeJIT represents code changes by using CTG. It first generates CPG of the code before and after the change, and then merges it, treating common nodes as unchanged nodes and added/deleted nodes as change nodes. Finally, in order to remove parts unrelated to code changes, all the nodes associated with change nodes are retained to form CTG.

Step2 (Feature Engineering): CodeJIT uses Word2vec [18] to construct node content vectors to capture semantic relations between code tokens. These vectors are further enhanced to form node feature vectors by integrating change operators (added, deleted, and unchanged). This integration is achieved by concatenating the respective one-hot vectors of the operators with the embedded vectors [15].

Step3 (JIT vulnerability detection with RGCN): CodeJIT feeds the embedded graph into a relational graph neural network. At each GNN layer, nodes exchange information with neighboring nodes via various relations to capture the local graph structure. After multiple GNN layers, a graph-level vector is formed by aggregating node features using a graph readout function (e.g., sum, average, max). Finally, a Multilayer Perceptron (MLP) is used to classify graph-level vectors, identifying vulnerabilities in graph.

However, there exist the following limitations.

Limitation 1 (Imprecise Feature Representation): Leveraging CPG has proven effective in capturing complex vulnerability features, which is why CodeJIT integrates the CPGs of both pre- and post-change files to construct a CTG. During this process, it attempts to prune nodes that are not structurally or semantically related to the changed code. However, the CPGs can grow exponentially as the lines of code increase, leading to

[2]https://github.com/mxzhou666/HgtJIT

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

a dramatic increase in the number of nodes. While pruning is intended to reduce noise, the strategy of including *all* contextual nodes which are related by structural or dependency relation can introduce a substantial amount of irrelevant or redundant information. This excessive inclusion creates a dense and noisy graph, which can decrease the learning capacity of GNN. As noted in [17], GNN-based models exhibit a marked decline in performance when the CPG of an input function surpasses 50 nodes. This performance degradation is more severe in the CTG, which are derived from entire files and therefore contain even more nodes. The sheer volume of nodes not only complicates the learning process but also increases the likelihood of the model missing critical vulnerability features due to the overwhelming amount of unrelated information.

Limitation 2 (Static Code Embedding): CodeJIT employed the Word2Vec technology trained on project-specific datasets to generate vector representations of source code for model training. However, such static code embeddings have been proven to be susceptible to the problem of OOV, leading to poor generalization ability [19]. In addition, programming languages are structured languages with unique syntactic and semantic structures. These factors are critical to accurately understanding the meaning of code and correctly representing its features. If code is simply treated as natural language, Word2vec may not be able to fully learn and utilize programming language domain-specific knowledge, much less absorb the code change domain knowledge hidden in code commits.

Limitation 3 (Underutilized Heterogeneous Information): CodeJIT utilizes CTG as input to relational graph neural networks for learning vulnerability features. However, the code change graph as a heterogeneous graph not only contains types information of edge (e.g., data dependencies, control dependencies) but also includes types information of node (e.g., addition and deletion of code change). Relational graph neural networks simply distinct non-sharing weights for edge type alone, making the model unable to jointly learn the semantic relations of edges and change information of nodes in the graph [20], [21]. In addition, CodeJIT includes node type information as part of node features, which weakens the model's focus on change information, leading to the model being unable to be aware of the semantic change relations brought about by the change information and affecting the learning of vulnerability features.

The limitations of CodeJIT motivate us to seek a more effective JIT vulnerability detection approach named HgtJIT. HgtJIT can build more accurate code change graphs, generate more meaningful node feature representations, and sufficiently utilize heterogeneous information of graph learning vulnerability features.

B. Code Change Representation Learning

In the field of JIT vulnerability detection, precise representation of code changes is critical. Code changes include operations such as addition, deletion, and modification, often spanning across multiple lines of code and files. Previous approaches [26] typically relied on manually designed features or rules for feature extraction to represent code changes as feature vectors. However, such approaches are confined to superficial features and struggle to capture the semantic information embedded within these changes.

In recent years, there has been substantial attention directed towards the technique of code change representation learning [27], [28], [29], [30], [31]. Its objective is to acquire a distributed representation of code changes, capturing higher-level features with semantic information, thereby better reflecting the content and intent of code changes. This technique has proven successful in various software engineering tasks, including commit message generation [32], [33], JIT defect prediction [24], [34], [35], and JIT comment update [36].

CCT5 [23] stands as a state-of-the-art pre-training model specifically focused on code change representation learning. It has been trained on a large-scale corpus of code changes, designed with five pre-training tasks specific to code changes. Its architecture is rooted in the T5 model [37], [38], employing an encoder-decoder framework comprising 12 layers of transformers, each utilizing 12 attention heads for multi-head attention computation. Utilizing self-attention mechanisms enables it to aggregate different parts of the token sequence with varying weights. Generally, attention weight calculation is as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (1)$$

Here, Q represents the query vector, K represents the key vector, V represents the value vector, and d_k is the dimension of the vectors.

In this study, we leverage CCT5 to embed graph nodes of diff-PDG, utilizing the domain knowledge of code changes learned on large-scale code datasets to generate the most meaningful feature representation.

C. Heterogeneous Graph Transformer

Heterogeneous graphs hold significant value in code analysis as they can consider multiple types of nodes and edges together to provide a more comprehensive reflection of the code's structure and semantics. In JIT vulnerability detection, heterogeneous graphs can be employed to represent the complexity of code changes, including additions, deletions, unmodified operations, and the dependency relations between them. However, effectively utilizing heterogeneous graphs poses a challenging task. Traditional graph neural network approaches mainly deal with homogeneous graphs and offer limited support for heterogeneous graphs. Hence, it is crucial to utilize the information of heterogeneous graphs to learn vulnerability features in JIT vulnerability detection.

HGT [20] is an emerging deep learning model designed specifically to process heterogeneous graph data with different node and edge types. Its idea is to parameterize the weight matrix of heterogeneous mutual attention, message passing, and propagation steps using the meta-relation of heterogeneous graphs. The meta-relation considers the types of edges along with their source and target nodes together. For a specific edge $E = (s, t)$ connecting nodes s and t, its meta-relation is expressed as $\tau(s)$, $\phi(e)$, $\tau(t)$, where τ and ϕ indicate the types of s and t, and e, respectively. This meta-relation characterizes the connectivity pattern within the heterogeneous graph. Understanding these

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

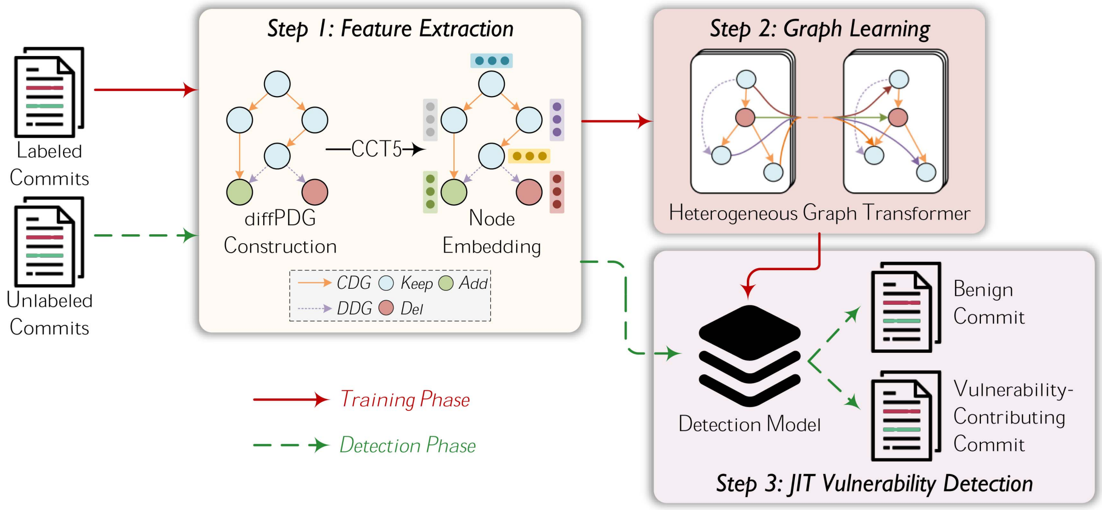


Fig. 1. The workflow of HgtJIT.

relations is crucial as they potentially contribute to the creation of more precise representations compared to considering node or edge types in isolation.

Currently, HGT has been applied to various domains, such as program analysis [21] and bioinformatics [39].

III. Methodology

A. Overview

To address the three key limitations of CodeJIT, we propose a JIT vulnerability detection approach called HgtJIT based on HGT, as shown in Fig. 1. First, we use PDG instead of CPG to generate code graph representation. We extract the PDG of the code before and after and merge them, mark common nodes as keep nodes, and mark new and deleted nodes as change nodes. Then, we perform forward and backward slicing based on these change nodes to preserve the context nodes strongly related to the change code to construct the diffPDG. This graph represents a greater emphasis on the dependency relations between codes while preserving the AST information in each statement node (the root of the AST). Second, we use the CCT5 model to generate the embedding representation of the graph nodes. CCT5 is pre-trained on a large-scale change dataset and able to generate more meaningful vector representations that retain more code semantics and change information compared to Word2vec trained on a specific limited dataset. Third, instead of using a relational graph neural network, we adopt HGT for heterogeneous graph learning. HGT utilizes its internal multi-head self-attention mechanism to focus on different nodes and relation types and fully leverages heterogeneous information to learn graph features. Finally, after processing through a classification layer, we obtain the commit vulnerability detection results. We will describe the technical details in the following chapters.

B. Feature Extraction

1) diffPDG Construction: In this work, We propose a novel change-level graph representation named diffPDG. The diffPDG can be constructed in three steps, namely, PDG construction, merge PDG, and slice to diffPDG. The details are as follows:

Generating Pre- and Post-Change PDGs: For each commit, it typically involves code changes both before and after. We employ the Joern [16] tool to parse the source code before and after the commit separately, generating the corresponding PDG denoted as $G = (V, E)$. Here, V represents the set of nodes, with each node representing a program statement in the code. E represents the set of edges, with each edge denoting data dependencies or control dependencies between code elements.

Merging into diffPDGs: In this phase, we merge the pre-change and post-change PDG into a unified representation, namely diffPDG. The primary process is as follows:

- To label nodes present in both pre-change and post-change states, representing unchanged code elements, i.e., keep nodes.
- To label nodes that exist in the post-change state but not in the pre-change state, signifying added code elements, i.e., add nodes.
- To label nodes that are present in the pre-change state but are absent in the post-change state, indicating deleted code elements, i.e., del nodes.

To avoid conflicts in node and edge identifiers during the merge, we reassign their identifiers. The merged graph can be represented as $G_{merge} = (V_{merge}, E_{merge})$, where V_{merge} and E_{merge} can be defined as:

$$V_{merge} = \{(id, code, vtype)|vtype \in \{keep, add, del\}\}$$
$$E_{merge} = \{(id1, id2, etype)|etype \in \{CDG, DDG\}\} \quad (2)$$

Here, id in V_{merge} denotes the unique identifier of a node, $code$ represents the content of the node, and $vtype$ signifies its change type. In E_{merge}, $id1$ and $id2$ represent the starting and ending nodes, and $etype$ denotes the dependencies between code elements.

Slicing diffPDGs: The slicing operation is a crucial step in the graph representation process. Its primary goal is to extract nodes and edges relevant to code changes from the merged PDG

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

to construct the diffPDG. During this process, we select changed code nodes (i.e., add nodes and del nodes) as the slicing criteria. The outcome of slicing is a set of nodes and edges directly or indirectly related to the slicing criteria. Slicing operations are based on control dependency and data dependency relations, including two directions of slicing: backward slicing and forward slicing [40].

To capture all relevant code nodes associated with source code nodes, we may conduct multiple slicing iterations. In each iteration, we consider code nodes already sliced and further trace their control and data dependency relations. This iterative process can be conducted in multiple rounds as needed, to ensure that we capture all potentially relevant code nodes associated with source code nodes.

Fig. 2(b) presents the constructed diffPDG for Fig. 2(a), where the solid/dotted edges denote control/data dependencies and red/green/blue circulars represent deleted/added/keep statements. Here, we set the number of iterations to 1 so that all the retained keep nodes are directly dependent on the changed nodes. Some nodes and edges are omitted for ease of viewing, respectively.

2) Node Embedding: To feed diffPDG into the HGT model, the node content in the graphs should be embedded into numeric vectors. we use CCT5 to embed the graph nodes of the diffPDG. CCT5 is the latest pre-trained model in code change representation learning. It allows us to combine large-scale code change knowledge with diffPDG to improve the performance of JIT vulnerability detection. The embedding process can be divided into the following two steps:

Tokenizer: Similar to the typical embedding process for code pre-trained models [38], [41], we begin by tokenizing the content of each node in the diffPDG using the RoBERTa tokenizer [42]. Each tokenized node is then represented as a token sequence, with a special $[CLS]$ token added at the beginning of the sequence. This can be represented as follows:

$$[CLS], c_1, c_2, ..., c_n \tag{3}$$

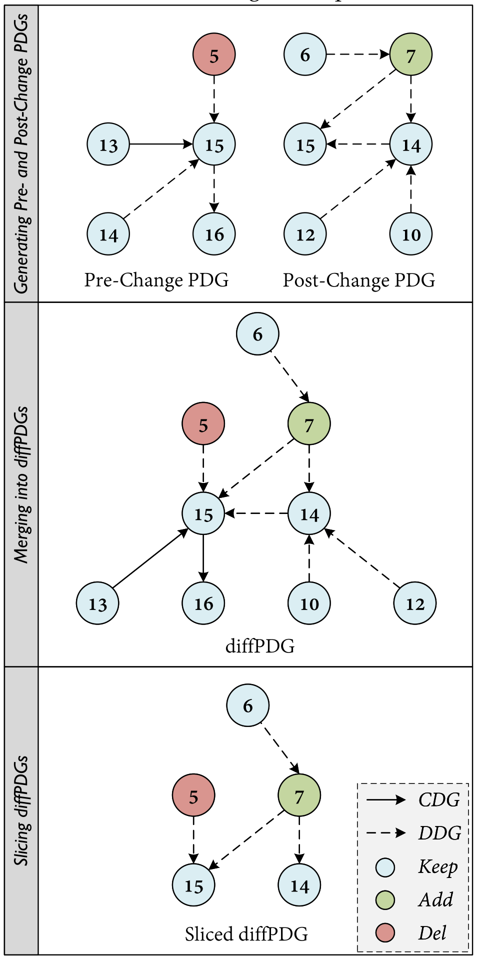
Here, $[CLS]$ serves as a special token that aggregates information from the entire token sequence, where c_i represents each token after tokenization.

CCT5 Encoder for Graph Node Embedding: The token sequences resulting from tokenization are input into the CCT5 model's encoder. CCT5 employs self-attention mechanisms, for the $[CLS]$ token, it participates in attention calculations with other tokens and generates attention weights. This means that $[CLS]$ establishes a connection with each token of the entire token sequence, which will carry the information of the entire token sequence and become an embedded representation of each statement node in the graph.

In this work, we do not use special tokens for different line types (e.g., $[ADD]$, $[DEL]$) because our goal is to utilize the CCT5 model to embed node content, capturing the semantic information in code changes, rather than using it as our final representation method. The diffPDG already effectively represents these changes information, and the HGT model will leverage this information for feature learning.

```
1    static int foo(AVBitStreamFilterContext *bsfc, AVCodecContext
2                   *avctx, const char *args, uint8_t **poutbuf,
3                   int *poutbuf_size, const uint8_t *buf,
4                   int buf_size, int keyframe){
5  -     int amount = args ? atoi(args) : 10000
6        unsigned int *state = bsfc ->priv_data;
7  +     int amount = args ? atoi(args) : (*state % 10001+1);
8        int i;
9
10       *poutbuf= av_malloc(buf_size + FF_INPUT_BUFFER_PADDING_SIZE);
11
12       memcpy(*poutbuf, buf, buf_size + FF_INPUT_BUFFER_PADDING_SIZE);
13       for(i=0; i<buf_size; i++){
14            (*state) += (*poutbuf)[i] + 1;
15            if(*state % amount == 0)
16               (*poutbuf)[i] = *state;
17       }
18       return 1;
19   }
```

(a) Code change example.



(b) The construction process of diffPDG of 2a.

Fig. 2. Example of diffPDG.

C. Graph Learning

We employ the HGT model, leveraging its ability to effectively handle heterogeneous graphs to better learn the vulnerability features within code changes. Fig. 3 shows the architecture of HGT, which aims to aggregate information from source node s using meta-relation to obtain a contextualized representation of target node t. This process can be decomposed into three parts: heterogeneous mutual attention, heterogeneous message passing, and target-specific aggregation. For ease of understanding, we will use a diffPDG subgraph as input to elaborate on important components in detail. Note, we use $H^{(l)}[t]$

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

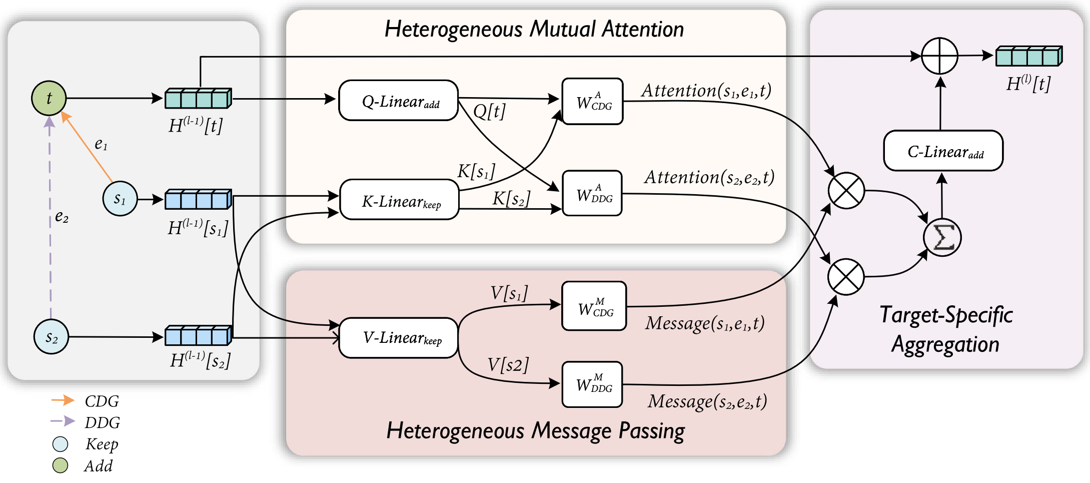


Fig. 3. The Architecture of a HGT layer.

to denote node t's embedding in the l-th layer, and $H^{(l-1)}[t]$ to denote node t's embedding $(l-1)$-th layer, through the whole section.

Heterogeneous Mutual Attention: The heterogeneous mutual attention phase determines the importance of heterogeneous messages during message passing, similar to a self-attention mechanism [43]. This process calculates the attention distribution of neighboring node s to node t in the graph while considering the node and edge types. For a specific edge $e = (s, t)$, the unnormalized attention score is computed using the formula below, taking into account both the node and edge types:

$$A^{(i)}(s,e,t) = \left(K^{(i)}(s) \cdot W^A_{\phi(e)} \cdot \left(Q^{(i)}(t)\right)^T\right) \cdot \frac{\mu_{\langle\tau(s),\phi(e),\tau(t)\rangle}}{\sqrt{d}} \tag{4}$$

In this formula, $Q^{(i)}_{\tau(t)}(t)$ and $K^{(i)}_{\tau(s)}(s)$ are the query and key obtained through the linear projection layers respectively for calculating attention, formulated as:

$$Q^{(i)}(t) = Q - Linear^{(i)}_{\tau(t)}(H^{(l-1)}[t]) \tag{5}$$

$$K^{(i)}(s) = K - Linear^{(i)}_{\tau(s)}(H^{(l-1)}[s]) \tag{6}$$

where i $(i \in [1,h])$ represent the i-th head of attention. The trainable prior variable $\mu_{\langle\tau(s),\phi(e),\tau(t)\rangle}$ acts as an adaptive scaling factor for each meta-relation triplet, where τ and ϕ indicate the types of s and t, and e, respectively. As shown in Fig. 3, there is a meta-relation $(keep, CDG, add)$ between s_1 and t, and there is a meta-relation $(keep, DDG, add)$ between s_2 and t. Nodes s_1 and s_2 both generate $K[s_1]$ and $K[s_2]$ through linear projection $K - Linear_{keep}$, while node t generates $Q[t]$ through $Q - Linear_{add}$. Subsequently, based on different edge types, $K[s_1]$ and $Q[t]$ are calculated for similarity through W^A_{CDG}, while $K[s_2]$ and $Q[t]$ are calculated for similarity through W^A_{DDG}.

After obtaining the unnormalized scores, a softmax activation is applied to them, and the results from multiple attention heads are concatenated to form the heterogeneous mutual attention $Attention^{(l)}(s,e,t)$, formulated as:

$$Attention^{(l)}(s,e,t) = \underset{i\in[1,h]}{\|} A^i(s,e,t) \tag{7}$$

After such processing, even between pairs of the same node type, the model can capture different semantic relations using meta-relation.

Heterogeneous Message Passing: HGT's message passing process is similar to the traditional Transformer model. For each edge e from node s to node t, heterogeneous messages are passed from s to t. Specifically, information from node s (denoted as $H^{(l-1)}[s]$) is first projected into the message space using $\mathrm{M} - \mathrm{Linear}_{\tau(s)}$, taking into account the node's type $\tau(s)$, and then incorporating the edge's type $\phi(e)$ dependency. The formula is as follows:

$$M^{(i)}(s,e,t) = M - Linear^{(i)}_{\tau(s)}\left(H^{(l-1)}[t]\right) \cdot W^M_{\phi(e)} \tag{8}$$

As shown in Fig. 3, s_1 and s_2 obtain $V[s_1]$ and $V[s_2]$ through the same linear projection $V - Linear_{keep}$, respectively. However, due to different meta-relations, they dot products with different W^M_{CDG} and W^M_{DDG}, multiple message heads independently generate messages, ultimately forming the heterogeneous message $Message^{(l)}(s,e,t)$, formulated as:

$$Message^{(l)}(s,e,t) = \underset{i\in[1,h]}{\|} M^i(s,e,t) \tag{9}$$

Target-Specific Aggregation: After collecting messages from all neighboring nodes $s \in \mathcal{N}(t)$ of each node t and calculating attention scores, we use these attention scores as weights to simply average messages and generate aggregated information for specific nodes. The aggregation formula is as follows:

$$a^{(l)}_t = \sum_{s\in\mathcal{N}(t)} \left(\mathrm{Attention}^{(l)}(s,e,t) \cdot \mathrm{Message}^{(l)}(s,e,t)\right) \tag{10}$$

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

Finally, the aggregated information of the node t (denoted as $a_t^{(l)}$) is combined with its previous residual information $H^{(l-1)}[t]$. This step is essential for the model to capture changes and updates, enriching node representations. The combination formula includes an activation function σ and $C - Linear_{\tau(t)}$, considering the node's type $\tau(t)$:

$$H^{(l)}[t] = \sigma \left(\text{C-Linear}_{\tau(t)} \left(a_t^{(l)} \right) \right) + H^{(l-1)}[t] \tag{11}$$

As shown in Fig. 3, after target-specific aggregation processing, we generate a new representation of the target node t in the current HGT layer $H^l[t]$. By overlaying L layers, we can obtain the node representation of the entire graph H^l.

D. JIT Vulnerability Detection

After using HGT to aggregate node information, we employ global attention pooling [44] to aggregate the representations of all nodes into a comprehensive representation of the entire diffPDG. Finally, leveraging a multi-layer perceptron (MLP) as a classifier, the generated graph representation is fed into the classifier for JIT vulnerability detection tasks, ultimately producing predictive outcomes.

IV. Experiments

A. Research Questions

To evaluate our proposed approach, we aim to address the following **Research Questions (RQs)**:

RQ1. Effectiveness on JIT Vulnerability Detection: To what extent can the JIT vulnerability detection performance HgtJIT achieve?

Recently, a state-of-the-art JIT vulnerability detection approach called CodeJIT has been proposed, However, as mentioned in Section II-A, CodeJIT has three key limitations, leading to inaccurate detection. Therefore, we propose our HgtJIT approach to address these challenges. We investigate if the performance of our HgtJIT outperforms the state-of-the-art JIT vulnerability detection approaches.

RQ2. Effectiveness in the cross-project setting: To what extent can HgtJIT maintain its vulnerability detection performance in the cross-project setting?

Cross-project evaluation is crucial to assess the generalization ability of JIT vulnerability detection models. We investigate whether our HgtJIT approach can effectively detect VCCs across projects and outperform the state-of-the-art approaches.

RQ3. Impact of diffPDG: How does the diffPDG impact the performance of HgtJIT?

HgtJIT uses diffPDG to represent code changes. We investigate whether diffPDG which focuses more on dependency relations contributed to JIT vulnerability detection compared to CTG. We also evaluate the impact of different context depths on the performance of HgtJIT.

RQ4. Impact of CCT5: How does generating feature representations using CCT5 impact the performance of HgtJIT?

HgtJIT uses a model CCT5 pre-trained on a large-scale code change dataset to embed graph nodes of diffPDG. We investigate whether feature representations with code change domain knowledge are effective in improving the JIT vulnerability detection performance of HgtJIT compared to other embedding approaches.

RQ5. Effectiveness of HGT: How does the HGT affect the performance of HgtJIT?

One of the key contributions of our approach is learning vulnerability features using HGT, which jointly learns the semantic relations of edges and change information of nodes in the diffPDG. We aim to show whether heterogeneous information in diffPDG captured by HGT contributes to JIT vulnerability detection in comparison with other popular GNNs.

TABLE I
Statistics of the Studied Dataset

	#Projects	#VCCs	#Non-VCCs	#Total	#Ratio
Original	FFmpeg	3,462	4,449	7,911	1:1.29
	QEMU	3,183	3,551	6,734	1:1.12
	Linux	780	783	1,562	1:1
	503 projects more...				
	Total	8,975	11,299	20,274	1:1.26
Filtered	FFmpeg	1,821	4,000	5,821	1:2.2
	QEMU	1,178	2,498	3,676	1:2.12
	Linux	271	596	867	1:2.2
	201 projects more...				
	Total	3,474	7,510	10,984	1:2.16

B. Datasets

We used the dataset provided by CodeJIT [15] for model training and evaluation. They used the SZZ algorithm [45] to label the commits that last modified the statements associated with vulnerabilities as VCCs. As shown in Table I, the CodeJIT dataset contained 20,274 commits including 11,299 non-VCCs and 8,975 VCCs (Column 3-5) from the vulnerabilities reported from Aug 1998 to Aug 2022 in real-world 506 C/C++ projects (Column 2) such as FFmpeg, QEMU, and Linux. Column 6 denotes the ratio of VCCs in each project.

We also performed dataset cleaning to filter (1) multi-file commits (i.e., a single commit involves changes to multiple files) because diffPDG was generated from a single file, and (2) commits which were mislabeled or could not be parsed. Such process was independently annotated by two authors. When 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100% of the dataset had been annotated, we conducted a cross-verification of their results. Cohen's Kappa coefficient rating aggregation was utilized to measure the discrepancy between the two annotators' labels [46]. There were a total of 11 rounds of labeling, with each round incorporating the dataset from the previous one. The Kappa coefficient ranges from 0.0% to 100.0%, where a higher value indicates better consistency. Generally, a Kappa coefficient exceeding 80% signifies a high degree of agreement between the two evaluations. In our labeling process, the overall Kappa coefficient achieved was 92.44%, indicating excellent consistency. After filtering, a total of 10,984 commits were retained, including 3,474 VCCs and 7,510 non-VCCs.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

C. Baselines

To evaluate our approach, we compared HgtJIT with the following five state-of-the-art JIT vulnerability detection approaches:

- *VCCFinder [12]:* A machine learning approach which uses commit messages and expert features to train a Support Vector Machine to identify VCCs.
- *DeepJIT [24]:* A CNN-based end-to-end deep learning model that automatically extracts features from commit messages and code change sets and predict whether a given commit is buggy or not.
- *CC2Vec [25]:* CC2Vec is a pre-trained model in the code change domain that generates distributed representations of code changes through a hierarchical attention network. It models the structural information of code changes and uses attention mechanisms to identify important aspects of code changes. Using CC2Vec to capture commit features and combining it with DeepJIT can achieve better results.
- *CCT5 [23]:* As the latest pre-trained model in the field of code change, it can adapt to various downstream tasks, including JIT vulnerability detection, through fine-tuning.
- *CodeJIT [15]:* A novel code-centric JIT vulnerability detection approach. It considers source code changes induced by commits as the direct and decisive factors for assessing the risk of a commit. It designed a novel graph-based multi-view code change representation approach, linking changed code in commits with unchanged code. It developed a graph-based JIT vulnerability detection model to capture vulnerability features at the commit-level.

D. Evaluation Metrics

In our study, to comprehensively evaluate the performance of different models, we utilized common evaluation metrics [47], including $Precision$, $Recall$, $F1$ and AUC. $Precision$ measures the model's ability to correctly predict positives, while $Recall$ measures the model's ability to capture positive cases. It can be expressed as $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$, where TP represents the true positives, FP and FN represent false positives and false negatives, respectively. The $F1$ is the harmonic average of precision and recall, expressed as $F1 = \frac{2\times Precision\times Recall}{Precision+Recall}$. The AUC (Area Under the Curve) measures the model's ability to distinguish between positive and negative classes. Specifically, it represents the area under the Receiver Operating Characteristic (ROC) curve, where a higher AUC indicates better model performance.

E. Implementation

We employed the widely used Joern [16] tool to generate PDG. Our experiments were conducted on a server running Ubuntu 18.04 with an NVIDIA Tesla T4 16GB GPU. We implemented the models using the PyTorch[3] framework and DGL[4] library. In our experiments, the feature encoder consisted of four HGT layers, with embedding and hidden layer sizes set to 768. Each HGT layer used eight attention heads with a dropout rate of 0.2. We used Adam [48] for optimizing with 64 batch size and a learning rate of $1e$-4. We set the maximum number of epochs in our experiment as 30 and adopted an early stop mechanism to obtain the best parameters. For each approach, we repeated the experiment 10 times and the final evaluation result we adopted was the average of each measurement metric.

[3]https://pytorch.org/
[4]https://github.com/dmlc/dgl

TABLE II
TEMPORAL SPLIT: PERFORMANCE EVALUATION RESULTS COMPARED TO BASELINES

Approach	Precision	Recall	F1	AUC	p-value
VCCFinder	0.34	0.58	0.43	0.54	0.002
DeepJIT	0.30	0.60	0.40	0.70	0.002
CC2Vec	0.30	**0.62**	0.41	0.71	0.002
CCT5	0.33	**0.62**	0.43	0.74	0.002
CodeJIT*	0.55	0.42	0.48	0.65	0.002
HgtJIT	**0.61**	0.51	**0.55**	**0.83**	/

* Since we filtered commits that span multiple files, were mislabeled, or cannot be compiled, CodeJIT's performance in our experiment is worse than that reported in the original paper.
Precision = 0.61 (p-value = 5.36e-4).
Recall = 0.62 (p-value = 1.08e-3).
AUC = 0.83 (p-value = 7.36e-4).

V. Experimental Results

A. RQ1: Effectiveness on JIT Vulnerability Detection

Experimental Design: We consider five aforementioned baselines: VCCFinder [12], DeepJIT [24], CC2Vec [25], CCT5 [23], and CodeJIT [15]. These approaches all have been open-sourced, we directly use their official implementations. Besides, in order to comprehensively compare the performance among baselines and HgtJIT, we consider $Precision$, $Recall$, $F1$ and AUC as evaluation metrics and conduct experiments on the dataset constructed in Section IV-B. We follow the same temporal split strategy to build the training data and testing data from the original dataset as previous works do [12], [24], [34]. Specifically, we first sort all commits by their timestamp in ascending. Then, the top 80% of commits are treated as training data, while the rest 20% of commits are treated as test data. In total, the training/testing split in the number of commits for this setting is 9,071/1,813. We extract 100 VCCs from the training/testing (80/20) set for qualitative analysis and these data are not used for model training and testing. To check if the performance difference between a baseline model and HgtJIT is statistically significant, we applied the Wilcoxon signed-rank test (a statistical test method corresponding to paired sample T-test which can compare whether the difference between the average of two groups is significant or not in the case of small samples) at a 95% significance level on their performance of 10 times experiments.

Results: The evaluation results are reported in Table II and the best performances are highlighted in bold. According to the results, HgtJIT outperforms all state-of-the-art baseline approaches on performance metrics except $Recall$. In particular, HgtJIT obtains 0.61, 0.55 and 0.83 in terms of $Precision$, $F1$ and AUC, which improves baselines by 10.9%-103.3%, 14.6%-37.5% and 12.2%-53.7% respectively. Table II also presents the p-values (tested by the Wilcoxon signed-rank test) when comparing HgtJIT with the baselines in terms of $F1$. We can

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

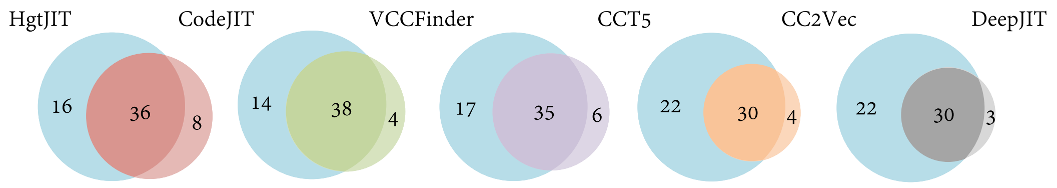


Fig. 4. Overlapping among results from our model and the baselines in RQ1.

observe that HgtJIT shows significant improvements (p-value < 0.05) over others. Fig. 4 show the overlapping analysis on the 100 extracted data from all of the models. HgtJIT discovers more unique VCCs than all other baselines. Specifically, 36, 38, 35, 30 and 30 of the true VCCs detected by CodeJIT, VCCFinder, CCT5, CC2Vec and DeepJIT, respectively, can also be detected by HgtJIT. Although CodeJIT, VCCFinder, CCT5, CC2Vec and DeepJIT can detect 8, 4, 6, 4 and 3 true VCCs that our model cannot detect, our model detects 16, 14, 17, 22 and 22 unique VCCs respectively.

Analysis: All these results demonstrate the effectiveness of our proposed HgtJIT in JIT vulnerability detection. As previous studies have shown [14], VCCFinder has lower performance due to inadequate code metrics. DeepJIT combines commit messages to enhance feature representations, but it has been shown that many commit messages are poor-quality [49], which can introduce noisy features. CC2Vec and CCT5 have high $Recall$ because they have learned code change knowledge from large-scale code corpus through pre-training, making it more sensitive to vulnerability features. However, they ignore the complex semantic relations in code changes, leading to a lack of contextual understanding. As a result, they fail to accurately identify vulnerability features, which causes lower $Precision$. Our approach HgtJIT utilizes diffPDG to represent code changes, fully considering the complex semantic relations between codes and achieving optimal $F1$ and AUC performance.

All metrics for CodeJIT are lower than HgtJIT. Because it uses Word2vec for node embedding and this feature representation is inadequate. In addition, although the powerful performance of relational graph neural networks in inferring potential vulnerability semantics from the code change graph makes CodeJIT outstanding, the underutilization of heterogeneous information still restricts the performance of CodeJIT in detecting more complex VCCs. In contrast, HgtJIT utilizes CCT5 to embed nodes, which fully utilize the domain knowledge in large-scale code change corpus to generate more meaningful feature representations and thus identify potential vulnerabilities more accurately. Besides, our approach jointly learns vulnerability features from node types and edge types in diffPDG and thus achieves better performance.

Answer to RQ1: HgtJIT excels in JIT vulnerability detection compared to state-of-the-art baselines, $F1$ and AUC improved by up to 37.5% and 53.7% respectively. It suggests that utilizing the heterogeneous information of code change graphs to learn vulnerability features is highly effective.

TABLE III
CROSS-PROJECT: PERFORMANCE EVALUATION RESULTS COMPARED TO JIT VULNERABILITY DETECTION BASELINES

Approach	Precision	Recall	F1	AUC	p-value
VCCFinder	0.32	**0.67**	0.43	0.49	0.002
DeepJIT	0.44	0.56	0.50	0.67	0.002
CC2Vec	0.47	0.56	0.51	0.70	0.002
CCT5	0.50	0.59	0.54	0.72	0.002
CodeJIT	**0.74**	0.50	0.60	0.73	0.011
HgtJIT	0.62	0.64	**0.63**	**0.80**	/

Precision = 0.74 (p-value = 4.44E-4).
Recall = 0.67 (p-value = 1.24E-3).
AUC = 0.80 (p-value = 8.09E-4).

B. RQ2: Effectiveness in the cross-project setting

Experimental Design: We explore how HgtJIT performs in the cross-project setting, i.e. samples from the target project do not appear in the source projects that are used in model training. Table I shows the distribution of the dataset by project. Specifically, all commits in 80% of the randomly divided projects are used to train the model, and commits in the remaining 20% of the projects are used to test. The training/test split in the number of commits for this setting is 9,982/1,002. We used the same baseline, metrics as in RQ1 and also performed the Wilcoxon signed-rank test.

Results: The evaluation results are reported in Table III and the best performances are highlighted in bold. HgtJIT obtains 0.63 and 0.80 in terms of $F1$ and AUC, which improves baselines by 5.0%-46.5% and 9.6%-63.3% in terms of $F1$ and AUC, respectively. Table III also shows HgtJIT significant improvements (p-value < 0.05) over others.

Analysis: All results indicate that HgtJIT remains effective in a cross-project setting. VCCFinder achieves the highest $Recall$ because code metrics exhibit similar feature distributions across different projects, leading the model to predict more samples as vulnerabilities. However, this also results in increased false positives and lower $Precision$. CodeJIT demonstrates the highest $Precision$ but the lowest $Recall$ among all approaches, it can be attributed to its inclusion of AST nodes in the graph, which provides a more fine-grained representation that enables more precise detection of vulnerabilities. In the cross-project settings, our $F1$ and AUC are the highest, suggesting that HgtJIT offers the best overall performance. This is valuable in real-world scenarios. HgtJIT exhibits strong generalization capabilities, enabling it to detect VCCs both effectively and accurately.

Answer to RQ2: HgtJIT still achieves optimal overall performance in the cross-project settings, with F1 and AUC

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

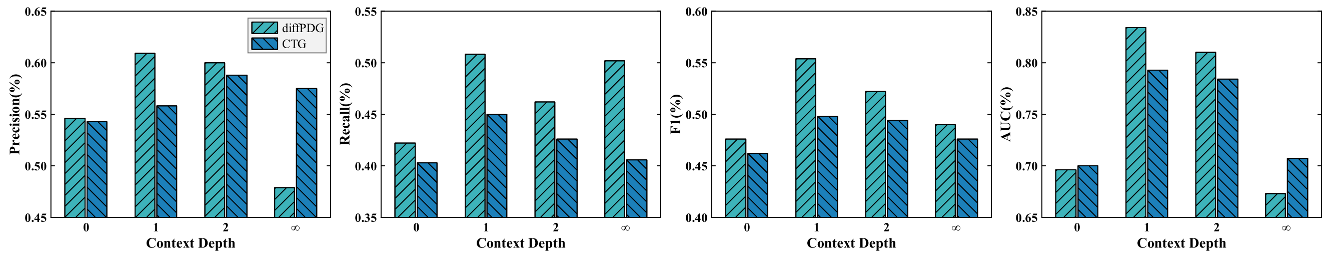


Fig. 5. Impact of diffPDG.

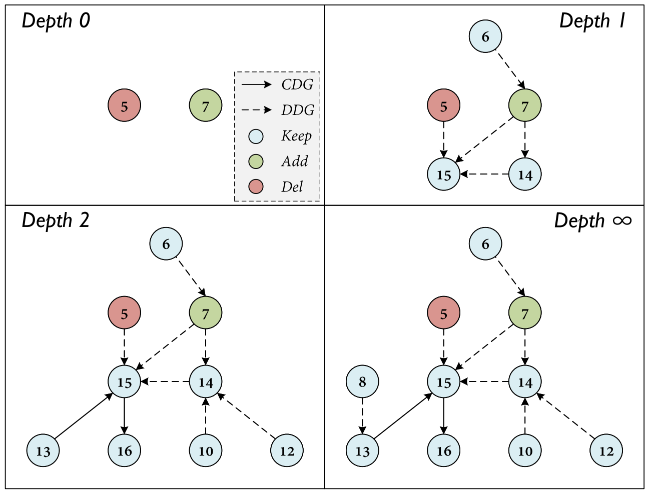


Fig. 6. Different depths of diffPDG sliced according to Fig. 2(a).

improved by up to 46.5% and 93.3%, indicating that HgtJIT has stronger generalization capabilities.

C. RQ3: Impact of diffPDG

Experimental Design: We generated the CTG strictly following the approach elaborated in the original paper, utilizing the Joern tool [16] to generate CPG to avoid deviations from the CodeJIT. In addition, contexts of different depths are preserved by slicing the graph based on the reachability from the change statements, considering K-hop neighboring nodes in each iteration. For a fair comparison with CTG, we also preserve different depths of context for CTG according to our slicing rules. We evaluated four different context depths. As shown in Fig. 6, Depth 0 means it contains only change statements, Depth 1 means it contains change statements and their direct context statements, Depth 2 means it contains change statements, direct context statements, and the nearest indirect context statements, and Depth ∞ means it contains change statements and all statements with direct and indirect contexts. We follow the same training and testing dataset in RQ1 for evaluation.

Results: The evaluation results as shown in Fig. 5. According to the results, we find that the model using diffPDG achieves optimal performance when considering only direct context statements, $Precision$, $Recall$, $F1$ and AUC with other depths are improved by 1.5%-27.1%, 1.2%-20.4%, 6.1%-15.4% and 3.0%-23.9% respectively. In addition, the model using diffPDG performs much better than the model using CTG at each context depth.

Analysis: All these results demonstrate the effectiveness of diffPDG in representing code changes in JIT vulnerability detection. Compared with no context (Depth 0 of diffPDG), direct context can facilitate vulnerability detection by complementing more semantic information. The performance gets worse when Depth is greater than 1, since indirect context may introduce too much noise and provide limited valuable information. The $F1$ drops by 6.1% when the Depth is 2, indicating that the indirect context does not carry much relevance for the changed code. The $F1$ further drops by 6.5% when considering all the indirect context (Depth ∞ of diffPDG), because most of these context statements are barely correlated with the changed statements but introduce excessive noise that significantly interferes with the detection model. The model using CTG shows a similar trend, obtaining optimal performance when the Depth is 1. The large amount of AST information in CTG models leads to overlooked dependency relations. In contrast, diffPDG models pay more attention to dependencies, which helps to detect vulnerabilities(as demonstrated in Section VI-A).

Answer to RQ3: The model using diffPDG performs significantly better than the model using CTG, indicating that focusing more on dependencies helps to detect vulnerabilities. In addition, direct context provides the most valuable information for inference.

D. RQ4: Impact of CCT5

Experimental Design: We considered three additional embedding approaches: Word2vec, Word2vec+Expert Features and CodeT5. We used the Word2vec [18] trained on our dataset to generate vector representations of statement nodes. Additionally, recent studies [50], [51] show source code vulnerabilities correlate highly with some specific syntax characteristics. For instance, in the C language, the usage of pointers and arrays is more susceptible to attacks. Therefore, extracting keyword frequencies related to pointers from each node's code snippet would be highly effective. We concatenated 20 expert features(Code Statement Metadata (2), Identifier and Literal Features (7), Control Flow Features (3), Operator Features (4), API Features (4)) extracted from each code statement node with the vector generated by Word2vec. We also used CodeT5 [38] to explore the effectiveness of pre-training models, CodeT5 is an advanced

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
IMPACT OF CCT5

Embedding	Precision	Recall	F1	AUC
Word2vec	0.55	0.40	0.46	0.80
Word2vec+Expert Features	0.53	0.44	0.48	0.79
CodeT5	0.56	0.49	0.52	0.80
CCT5	**0.61**	**0.51**	**0.55**	**0.83**

Precision = 0.61 (p-value = 5.36e-4).
Recall = 0.51 (p-value = 2.72e-3).
F1 = 0.55 (p-value = 2.53e-3).
AUC = 0.83 (p-value = 7.36e-4).

pre-trained model in the code domain. We follow the same training and testing dataset in RQ1 for evaluation.

Results: Table IV shows the results of different embedding approaches. We observe that CCT5 effectively improves model performance. Compared to the other embedding approaches, $Precision$, $Recall$, $F1$ and AUC improved by 8.9%-15.1%, 4.1%-27.5%, 5.8%-19.6% and 3.8%-5.1% respectively.

Analysis: All results demonstrate that using CCT5 can improve the performance of JIT vulnerability detection. Word2vec has the worst performance. Adding expert features can add some vulnerability features, but the effect is still poor. In addition, the code feature representation generated by training based on a specific corpus is insufficient and has OOV problems. In contrast, CodeT5 is able to capture structural and semantic information of the code and generate more meaningful feature representations by training on a large-scale code corpus. However, all the performance metrics of CodeT5 are worse than CCT5, which is because CCT5 is a pre-trained model for the domain of code changes. In addition to the code syntactic and semantic information, it also understands the differences of code changes [23]. This be attributed to CCT5 designing specific pre-training tasks for code changes during the pre-training phase.

Answer to RQ4: CCT5 can effectively contribute to the performance of HgtJIT, as it can utilize the domain knowledge in large-scale code change corpus to generate more meaningful feature representations.

E. RQ5: Effectiveness of HGT

Experimental Design: Due to the outstanding ability in processing graph data structures, GNNs have been widely used in vulnerability detection and have achieved great breakthroughs. We respectively replaced the HGT model with three famous GNN models, including GCN (Graph Convolutional Network) [52], RGCN (Relation Graph Convolutional Network) [53], and RGAT (Relation Graph Attention Network) [54], to evaluate the contribution of each model to JIT vulnerability detection. The experimental dataset is set the same as the experiment of RQ1. Additionally, We set the hyperparameters according to the experience of the original paper.

Results: Table V shows the results of different GNNs. The results demonstrate that utilizing HGT as the model for learning vulnerability features achieves optimal performance. Compared to the other GNNs, $Precision$, $Recall$, $F1$ and AUC improved by 10.9%-32.7%, 6.2%-10.9%, 5.8%-19.6% and 18.6%-27.7%, respectively.

TABLE V
EFFECTIVENESS OF HGT

Model	Precision	Recall	F1	AUC
GCN	0.46	0.46	0.46	0.65
RGCN	0.53	0.46	0.49	0.66
RGAT	0.55	0.48	0.52	0.70
HGT	**0.61**	**0.51**	**0.55**	**0.83**

Precision = 0.61 (p-value = 5.36e-4).
Recall = 0.51 (p-value = 2.72e-3).
F1 = 0.55 (p-value = 2.53e-3).

Analysis: All results demonstrate the effectiveness of HGT in learning vulnerability features. We observe that the performance of GCN is poor. The main reason is that the neglection of edge types leads to the missing of structured code features (e.g., control- and data-flow). Without accurate control- and data-flow information, the performance of JIT vulnerability detection drops. RGCN aggregates node and edge information through the directed edge. RGAT uses the attention mechanism on multiple edge types, its precise learning of relations between nodes makes it achieve higher performance. Though they both consider inter-node relations, the neglect of node types makes it difficult to detect vulnerabilities caused by changes. While HGT can utilize the information of node type and relation type in heterogeneous graphs more comprehensively based on meta-relations to identify diversified vulnerabilities and achieve higher performance.

Answer to RQ5: The model trained with HGT achieves optimal performance, confirming the effectiveness of HGT which considers heterogeneous information in learning vulnerability features in code changes.

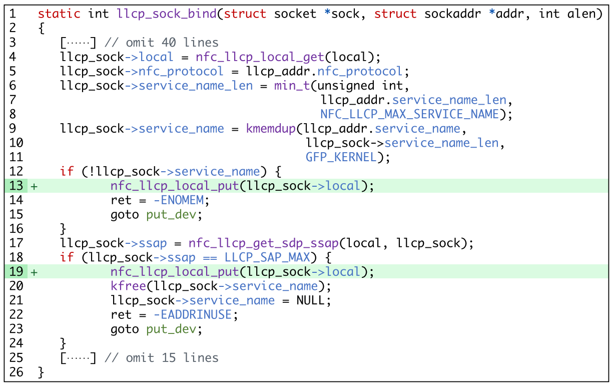
VI. DISCUSSION

A. Case Study

Fig. 7(a) illustrates an instance of vulnerability-contributing identified by our approach in the Linux kernel. Specifically, Commit $c33b1cc$[5] addressed a vulnerability identified as CVE-2020-25670.[6] In the function /texttttllcp_sock_bind() (at line 1), a reference to a local resource of the NFC protocol stack was obtained by calling function /texttttnfc_llcp_local_get(local) (at line 4). When memory allocation fails (at line 12) or SSAP reaches its maximum value (at line 18), the program returns an error code, but the function /texttttnfc_llcp_local_put() was not invoked, leading to the improper release of the previously acquired NFC protocol stack local resource, resulting in a refcount leak, which is a potential use-after-free

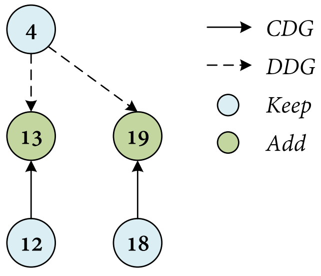
[5]https://github.com/chipcraft-ic/toolchain-component-linux/commit/c33b1cc62ac05c1dbb1cdafe2eb66da01c76ca8d
[6]https://nvd.nist.gov/vuln/detail/CVE-2020-25670

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.



```
1    static int llcp_sock_bind(struct socket *sock, struct sockaddr *addr, int alen)
2    {
3       [······] // omit 40 lines
4       llcp_sock->local = nfc_llcp_local_get(local);
5       llcp_sock->nfc_protocol = llcp_addr.nfc_protocol;
6       llcp_sock->service_name_len = min_t(unsigned int,
7                                           llcp_addr.service_name_len,
8                                           NFC_LLCP_MAX_SERVICE_NAME);
9       llcp_sock->service_name = kmemdup(llcp_addr.service_name,
10                                        llcp_sock->service_name_len,
11                                        GFP_KERNEL);
12      if (!llcp_sock->service_name) {
13 +            nfc_llcp_local_put(llcp_sock->local);
14              ret = -ENOMEM;
15              goto put_dev;
16      }
17      llcp_sock->ssap = nfc_llcp_get_sdp_ssap(local, llcp_sock);
18      if (llcp_sock->ssap == LLCP_SAP_MAX) {
19 +            nfc_llcp_local_put(llcp_sock->local);
20              kfree(llcp_sock->service_name);
21              llcp_sock->service_name = NULL;
22              ret = -EADDRINUSE;
23              goto put_dev;
24      }
25      [······] // omit 15 lines
26   }
```

(a) Commit *c*33*b*1*cc* to fix CVE-2020-25670 but introduce other vulnerabilities.



(b) diffPDG of 7a.

Fig. 7. Example of vulnerability-contributing

(CWE-416) vulnerability. To address this issue, a call to /texttt nfc_llcp_local_put(llcp_sock->local) (at line 13 and line 19) was added to properly release the previously obtained local resource. However, this fix contributed to a new vulnerability. Although calling function /texttt nfc_llcp_local_put() will release the relevant memory, /texttt llcp_sock->local still retains references to the released memory addresses. If the same local resource is assigned to two different sockets, it will result in a use-after-free vulnerability, identified as CVE-2021-23134.[7]

Overall, our approach identifies nodes with data dependencies (at line 4) and control dependencies (at line 13 and line 19) with added code by constructing a diffPDG of commit *c*33*b*1*cc* (as shown in Fig. 7(b)). Data dependencies can make our model aware of the counting issue of memory references involved in the commit, while control dependencies may make the model aware that references should be fully released in a timely manner when the program needs to return. We use the HGT model that is sensitive to heterogeneous information for vulnerability feature learning, this combination enables HgtJIT to provide profound insights into the root causes of vulnerability introductions, offering robust support for software security. The application of our approach in the Linux kernel demonstrates its feasibility and practicality in real-world software development environments.

B. LLMs for JIT Vulnerability Detection

1) LLM-based Just-in-Time Vulnerability Detection: Recent advances in large language models (LLMs) have demonstrated their strong capabilities across a wide range of software engineering tasks, such as code summarization, program repair, and bug detection. Motivated by this progress, we conduct a preliminary exploration of whether LLMs can be applied to the JIT vulnerability detection task.

[7]https://nvd.nist.gov/vuln/detail/CVE-2021-23134

TABLE VI
PERFORMANCE OF LARGE LANGUAGE MODELS APPLIED TO JIT VULNERABILITY DETECTION

Model	Precision	Recall	F1	AUC
GPT-4o-mini	0.19	0.79	0.31	0.52
GPT-3.5-turbo	0.19	**0.93**	0.32	0.51
Llama-4-scout	0.22	0.50	0.31	0.56
DeepSeek-V3	0.22	0.33	0.27	0.50
HgtJIT	**0.61**	0.51	**0.55**	**0.83**

Precision = 0.61 (p-value = 5.36e-4).
Recall = 0.93 (p-value = 4.88e-3).
F1 = 0.55 (p-value = 2.53e-3).
AUC = 0.83 (p-value = 7.36e-4).

To evaluate the effectiveness of LLMs for JIT vulnerability detection, we adopt a prompt-based zero-shot classification setting. Specifically, we construct a binary classification task, where an LLM is asked to predict whether a given code change introduces a vulnerability. The prompt is designed as follows:

LLM Detection Prompt Template

You are a software security expert with 10 years of experience in vulnerability analysis. Given a code change (diff), you need to analyze whether the code change will introduce potential vulnerabilities. If there are any potential vulnerabilities, strictly return 1. If not, strictly return 0. No further explanation is needed. The code is as follows:
-diff:{}

We evaluate four publicly available LLMs: GPT-4o-mini [55], GPT-3.5-turbo [56], Llama-4-Scout [57], and DeepSeek-V3 [58]. For consistency, all models are queried using the same prompt template and follow the same testing dataset and metrics in RQ1 for evaluation.

As shown in Table VI, all LLMs exhibit limited performance in the JIT vulnerability detection setting. For instance, GPT-4o-mini and GPT-3.5-turbo achieve high recall scores but suffer from extremely low precision, leading to a large number of false positives. This result shows that LLMs often treat most code changes as risky, even when they are safe. This could be because these models are trained to be cautious in general tasks, rather than being fine-tuned to understand what actually makes a code change vulnerable. While Llama-4-Scout performs slightly better than the others in terms of F1 score and AUC, its results are still far from what's needed for real-world use. In particular, all the AUC scores are close to 0.5, which means the models can barely tell the difference between truly vulnerable and safe code. These results suggest that LLMs are not yet suitable for accurate vulnerability detection during code commit. In contrast,

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

HgtJIT is specifically designed for this task, offers more stable and reliable predictions.

2) LLM-based Explanation of Detected Vulnerabilities: While our proposed HgtJIT can assist developers in detecting vulnerability-contributing commits, it would be more meaningful for developers if we could also offer explainable information. Existing works have shown the potential of LLMs in code understanding and security tasks [59], [60]. To investigate whether LLMs can provide explainable information for JIT vulnerability detection, we conducted an experiment focusing on explanation generation. Specifically, we designed a prompt that asks the LLM to identify the exact location of the vulnerability in a known vulnerability-contributing commit and provide a brief explanation. The prompt template is as follows:

LLM Explanation Prompt Template

You are a software security expert with 10 years of experience in vulnerability analysis. Given a known vulnerability contributing commit, you need to analyze the specific location where the vulnerability was introduced and provide a concise explanation of the vulnerability. The code is as follows:
-diff:{}

We selected 50 vulnerability-contributing commits from our dataset and used four popular LLMs: GPT-4o-mini, GPT-3.5-turbo, Llama-4-Scout, and DeepSeek-V3. To evaluate the quality of the generated results, we performed a lightweight manual assessment. Each explanation was independently reviewed by two experienced researchers in software security. The reviewers were asked to assess (1) whether the explanation correctly identified the vulnerable code location, and (2) whether the explanation provided was coherent and meaningful.

Each explanation was scored as 'useful' if it both pointed to vulnerable code and provided a reasonable explanation, 'partially useful' if it only achieved one of the two, and 'not useful' otherwise. We found that only 14% of GPT-4o-mini responses and 16% of GPT-3.5-turbo responses were completely useless, while Llama-4-Scout and DeepSeek-V3 were both 20% useless.

Interestingly, even with only diff information and no additional context, the LLMs were often able to locate risky code changes or give reasonable explanations. In many cases, explanations mentioned issues such as unchecked pointer use, missing validation, or unsafe API calls, which align well with the true root causes. For example, in the CVE sample shown in Table VII, the model explanation focuses on the field `llcp_sock->local` and highlights the risk of both null pointer dereference and use-after-free vulnerabilities. This aligns well with the actual cause described in the CVE. This suggests that, although LLMs may struggle with precise vulnerability classification, they can still serve as useful assistants for explanation. Such capability can complement models like HgtJIT, which provides accurate detection results but lacks explanation. Combining both could be more effective in preventing the introduction of vulnerabilities.

TABLE VII
LLM-Generated Explanation Case

CVE-2021-23134
CVE Description: Use After Free vulnerability in **nfc sockets** in the Linux Kernel before 5.12.4 allows local attackers to elevate their privileges. In typical configurations, the issue can only be triggered by a privileged local user with the CAP_NET_RAW capability.
LLM Explanation: The specific vulnerability introduced appears to stem from potential misuse of resource management pertaining to `llcp_sock->local`. **There may be a risk of both null pointer dereference and use-after-free issues if this locality is improperly handled elsewhere in the codebase** (e.g., if the object was already freed or not initialized appropriately). **Understanding how `llcp_sock->local` is managed throughout the lifecycle will be crucial to fully assess the impact of this change.**

C. Efficiency of HgtJIT

In terms of inference efficiency, HgtJIT takes approximately 1.2 seconds to classify a single commit. Compared to CodeJIT's reported classification time of 0.75 seconds (with FastRGCN), HgtJIT incurs slightly higher latency due to its more complex graph structure and multi-relational attention mechanism. We also observed that increasing the model depth or context scope moderately increases training time, but does not significantly affect inference time. For example, extending the transformer layers from 2 to 4 increases training time by about 1.6 times, while the average classification time remains stable at around 1.2 seconds per commit.

Overall, the HgtJIT's inference time is still acceptable for just-in-time feedback during code commits.

D. Threats to Validity

Threats to Internal Validity in our experiment relates to two factors. The first threat is the quality of our experimental datasets, we used the dataset provided by CodeJIT [15] and conducted further processing, as detailed in Section IV-B. However, noise label filtering exists artificial deviation. To mitigate this threat, two postgraduates and one Ph.D. participated in this process. We evaluate inter-annotator consistency by means of the kappa coefficient. If two postgraduates disagreed on the label of the same sample, the sample would be forwarded to the Ph.D. evaluator for further investigation. The second threat is the potential mistakes in the implementation of baselines. To minimize such a threat, we use the original source code of baselines from the GitHub repositories shared by corresponding authors and use the same hyperparameters in the original papers.

Threats to External Validity primarily concerned with the generalizability of our approach. We only conduct our experiments on C/C++ datasets, and thus our experimental results may not generalizable to other programming languages. However, the code graph representations we generate are not limited to specific programming languages, and Joern can perform code analysis for multiple programming languages such as Java and Python. As a result, our approach can be easily extended to other programming languages, an area we aim to explore in our future research.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

VII. Related Work

A. JIT Vulnerability Detection Based on Machine Learning

Traditional JIT vulnerability detection techniques typically use code commit metrics and text features as data inputs, leveraging machine learning models to identify potential vulnerabilities. For example, Perl et al. [12] introduced a unique approach that utilizes the "git blame" command in the Git version control system to trace modifications made in code commits that deleted lines associated with known vulnerability-fixing commits. The most frequently blamed commit in this process is labeled as a "VCC." Subsequently, they constructed a dataset containing these VCCs and trained machine learning models, such as Support Vector Machines (SVM), on this dataset. Their model demonstrated outstanding performance in detecting potential vulnerabilities, surpassing some static analysis tools, and highlighting the potential of machine learning in the field of vulnerability detection.

Riom et al. [61] replicated Perl et al.'s study and attempted to further enhance the vulnerability prediction model. They focused on exploring different feature sets, including those related to security aspects, such as the number of "sizeof" operators, which are associated with improper sizing of dynamically allocated buffers. However, it is worth noting that they faced challenges related to the unavailability of datasets and scripts, and the original paper did not provide sufficient replication details, making a strict comparison with Perl et al.'s research unfeasible.

Additionally, Yang et al. [13] concentrated on web vulnerabilities within Mozilla Firefox. They utilized an extensive set of process and product metrics to provide a high-precision JIT vulnerability prediction model. While their model excelled in terms of precision, the recall rate was relatively lower, implying that in the best configuration, the model might miss some potential vulnerabilities. These studies underscore the significance of machine learning in JIT vulnerability detection, particularly when leveraging version control system information to identify and predict potential vulnerabilities.

Unlike traditional machine learning methods, HgtJIT directly learns vulnerability features from the changed code. Through effective graph representation methods and model learning, we have achieved significant performance improvements.

B. Vulnerability Detection Based on Deep Learning

Vulnerability detection has been a critical concern in the field of software security. Traditional approaches for vulnerability detection primarily utilize software metrics, such as code complexity, as features to identify potential vulnerabilities. However, the manual collection of such software metrics is often time-consuming and labor-intensive. Consequently, in recent years, deep learning approaches have made significant strides in vulnerability detection, offering the capability to automatically learn patterns of vulnerabilities from historical data.

One prevalent deep learning architecture employed for this purpose is the Recurrent Neural Network (RNN), especially the Long Short-Term Memory (LSTM) network, to automatically acquire semantic and syntactic features from source code. For instance, Russell et al. [10] introduced an RNN architecture to automatically extract source code features for vulnerability prediction. Similarly, Dam et al. [6] proposed an LSTM-based architecture to automatically learn the semantic and syntactic features of source code. Nevertheless, these RNN approaches frequently assume that source code is a sequence of tokens, neglecting the graph structure inherent in source code, such as the Abstract Syntax Tree, potentially leading to inaccurate predictions.

In response to this challenge, Li et al. introduced VulDeePecker [62], an RNN-based model that learns from different graph properties of source code, such as the Data Dependency Graph. However, VulDeePecker still learns graph properties sequentially and does not make full use of GNNs. In recent years, several studies [63], [64] have begun to employ Graph Neural Networks to learn the graph properties of source code for vulnerability prediction. For example, Zhou et al. [11] utilized a Graph Neural Network to learn four types of graph properties of source code, including the Abstract Syntax Tree, Control Flow Graph, Data Flow Graph, and syntactic features. Additionally, Chakraborty et al. [5] introduced Reveal, a model that utilizes a Gated Graph Neural Network (GGNN) to learn the graph properties of source code.

Although several approaches to vulnerability prediction have been proposed, they have mainly focused on coarser levels of granularity, such as files, functions, and approaches. Consequently, Li et al. introduced VulDeeLocator [9], which utilizes program slicing techniques to narrow down the scope of vulnerability localization. Furthermore, Li et al. proposed IVDetect [8], which leverages GNNs for function-level predictions and utilizes GNNExplainer to identify the sub-graph contributing the most to predictions. Most recently, Fu et al. presented LineVul [7], a line-level vulnerability detection approach based on the Transformer architecture. Building upon IVDetect, LineVul achieves outstanding performance, marking a significant breakthrough in the domain of line-level vulnerability detection, providing enhanced precision for software security.

Recently, LLMs have emerged as a promising paradigm in vulnerability detection. Some approaches through prompt engineering guide LLMs to perform vulnerability detection either in zero-shot or few-shot settings. For instance, Zhou et al. [59] design prompts with task descriptions and vulnerable code examples to help LLMs better identify vulnerability patterns. Ni et al. [60] adopt few-shot prompting by providing several labeled code examples within the context window, allowing the model to learn from examples. To further enhance LLM performance, retrieval-augmented generation (RAG) techniques have been proposed. These approaches retrieve semantically similar examples or relevant domain knowledge to enrich the prompt. Zhou et al. [59] use CodeBERT to retrieve similar code snippets as guidance. Du et al. [65] construct a CVE-based knowledge base and retrieve functionally relevant vulnerability information to support detection and suggest potential fixes.

These approaches primarily focus on detecting vulnerabilities at the file- or function-level. While HgtJIT focuses on the commit-level, it can swiftly detect vulnerabilities at the time of code commit, fundamentally reducing the potential risks associated with VCCs.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

VIII. Conclusions and Future Work

In this paper, we propose an innovative approach designed to enhance the performance of JIT vulnerability detection. Our approach involves the construction of a diffPDG, utilizing CCT5 for graph node embeddings to capture code change information, and leveraging the HGT to effectively learn vulnerability features. Through experiments and performance evaluations, our approach offers a promising solution in the field of JIT vulnerability detection, with the potential for significant performance improvements in practical applications.

In the future, we aim to build higher-quality and more diverse datasets, including various projects in different programming languages, to further enhance the model's generalization capabilities. In addition, we plan to analyze multi-file commits to find an effective cross-file representation method to support the detection of such commits. Furthermore, we intend to collaborate with industry partners to integrate HgtJIT into large-scale codebases, enabling us to evaluate its practical effectiveness and refine it based on real-world deployment feedback.

References

[1] Flawfinder, 2024. [Online]. Available: http://www.dwheeler.com/FlawFinder

[2] Checkmarx, 2024. [Online]. Available: https://www.checkmarx.com

[3] Infer, 2024. [Online]. Available: https://fbinfer.com

[4] 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. 44th IEEE/ACM Int. Conf. Softw. Eng.*, 2022, pp. 1456–1468.

[5] 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.

[6] 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.

[7] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. 19th IEEE/ACM Int. Conf. Mining Softw. Repositories*, 2022, pp. 608–620.

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

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

[10] R. L. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl.*, 2018, pp. 757–762.

[11] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. 33rd Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 10197–10207.

[12] H. Perl et al., "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 426–437.

[13] L. Yang, X. Li, and Y. Yu, "VulDigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *Proc. 18th IEEE Glob. Commun. Conf.*, 2017, pp. 1–7.

[14] F. Lomio, E. Iannone, A. D. Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet," *J. Syst. Softw.*, vol. 188, 2022, Art. no. 111283.

[15] S. Nguyen, T. Nguyen, T. T. Vu, T. Do, K. Ngo, and H. D. Vo, "Code-centric learning-based just-in-time vulnerability detection," *J. Syst. Softw.*, vol. 214, 2024, Art. no. 112014.

[16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. 35th IEEE Symp. Secur. Privacy*, 2014, pp. 590–604.

[17] X. 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. 45th IEEE/ACM Int. Conf. Softw. Eng.*, 2023, pp. 2275–2286.

[18] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Representations*, 2013, pp. 1–12.

[19] S. Cao et al., "Learning to detect memory-related vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 43:1–43:35, 2024.

[20] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proc. 29th Web Conf.*, 2020, pp. 2704–2710.

[21] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proc. 30th IEEE/ACM Int. Conf. Prog. Comprehension*, 2022, pp. 378–389.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[23] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "CCT5: A code-change-oriented pre-trained model," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2023, pp. 1509–1521.

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

[25] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proc. 42nd Int. Conf. Softw. Eng.*, Seoul, 2020, pp. 518–529.

[26] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, May 2018.

[27] R. C. Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *SN Comput. Sci.*, vol. 2, no. 3, 2021, Art. no. 150.

[28] Z. Liu, Z. Tang, X. Xia, and X. Yang, "CCRep: Learning code change representations via pre-trained code model and query back," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, 2023, pp. 17–29.

[29] Z. Li et al., "Automating code review activities by large-scale pre-training," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2022, pp. 1035–1047.

[30] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 22:1–22:12.

[31] F. Zhang, B. Chen, Y. Zhao, and X. Peng, "Slice-based code change representation learning," in *Proc. 30th IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2023, pp. 319–330.

[32] J. Dong et al., "FIRA: Fine-grained graph-based code change representation for automated commit message generation," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 970–981.

[33] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 373–384.

[34] C. Pornprasit and C. Tantithamthavorn, "JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction," in *Proc. 18th IEEE/ACM Int. Conf. Mining Softw.*, 2021, pp. 369–379.

[35] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: Integrating semantic features with expert features for defect prediction and localization," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2022, pp. 672–683.

[36] B. Lin, S. Wang, Z. Liu, X. Xia, and X. Mao, "Predictive comment updating with heuristics and AST-path-based neural learning: A two-phase approach," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1640–1660, Apr. 2023.

[37] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.

[38] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. 26th Conf. Empirical Methods Natural Lang. Process.*, 2021, pp. 8696–8708.

[39] X. Mei, X. Cai, L. Yang, and N. Wang, "Relation-aware heterogeneous graph transformer based drug repurposing," *Expert Syst. Appl.*, vol. 190, 2022, Art. no. 116165.

[40] J. Cai, B. Li, J. Zhang, X. Sun, and B. Chen, "Combine sliced joint graph with graph neural networks for smart contract vulnerability detection," *J. Syst. Softw.*, vol. 195, 2023, Art. no. 111550.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

[41] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," 2020, *arXiv: 2002.08155*.
[42] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv: 1907.11692*.
[43] A. Vaswani et al., "Attention is all you need," in *Proc. 31st Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[44] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proc. 4th Int. Conf. Learn. Representations*, 2016.
[45] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.
[46] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, pp. 159–174, 1977.
[47] M. Pendleton, R. Garcia-Lebron, J. Cho, and S. Xu, "A survey on systems security metrics," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 62:1–62:35, 2017.
[48] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. 7th Int. Conf. Learn. Representations*, 2019, pp. 1–18.
[49] Y. Tian, Y. Zhang, K. Stol, L. Jiang, and H. Liu, "What makes a good commit message?," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 2389–2401.
[50] X. Wang, S. Wang, K. Sun, A. L. Batcheller, and S. Jajodia, "A machine learning approach to classify security patches into vulnerability types," in *Proc. 8th IEEE Conf. Commun. Netw. Secur.*, 2020, pp. 1–9.
[51] 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. Dependable Secur. Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul./Aug. 2022.
[52] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Representations*, 2017, pp. 1–14.
[53] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017, *arXiv: 1703.06103*.
[54] D. Busbridge, D. Sherburn, P. Cavallo, and N. Y. Hammerla, "Relational graph attention networks," 2019, *arXiv: 1904.05811*.
[55] GPT-4o, 2025. [Online]. Available: https://openai.com/
[56] ChatGPT, 2025. [Online]. Available: https://chatgpt.com/
[57] Llama, 2025. [Online]. Available: https://www.llama.com/
[58] DeepSeek, 2025. [Online]. Available: https://chat.deepseek.com/
[59] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proc. 46th ACM/IEEE 44th Int. Conf. Softw. Eng.: New Ideas Emerg. Results*, 2024, pp. 47–51.
[60] C. Ni, L. Shen, X. Xu, X. Yin, and S. Wang, "Learning-based models for vulnerability detection: An extensive study," 2024, *arXiv:2408.07526*.
[61] T. Riom, A. D. Sawadogo, K. Allix, T. F. Bissyandé, N. Moha, and J. Klein, "Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits," *Empir. Softw. Eng.*, vol. 26, no. 3, 2021, Art. no. 46.
[62] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. System Secur. Symp.*, 2018, pp. 1–15.
[63] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, 2021, Art. no. 106576.
[64] S. Cao et al., "Coca: Improving and explaining graph neural network-based vulnerability detection systems," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng.*, 2024, pp. 155:1–155: 13.
[65] X. Du et al., "VUL-rag: Enhancing LLM-based vulnerability detection via knowledge-level RAG," 2024, *arXiv:2406.11147*.

Xiaobing Sun (Member, IEEE) received the BS degree in computer science and technology from the Jiangsu University of Science and Technology, Zhenjiang, China, in 2007, and the PhD degree in computer science and technology from the School of Computer Science & Engineering, Southeast University, Nanjing, China, in 2012. He is a professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. He has been authorized more than 20 patents, and authored and co-authored more than 80 papers in referred international journals and conferences. His research interests include software maintenance and evolution, software repository mining, and intelligence analysis, etc.

Mingxuan Zhou received the BE degree in computer science and technology from the Yangzhou University, Yangzhou, China, in 2021. He is currently working toward the MSc degree with the School of Information Engineering, Yangzhou University, Yangzhou, China. His research interests include software security and deep learning.

Sicong Cao received the BE degree in software engineering from the Nanjing Institute of Technology, Nanjing, China, in 2019. He is currently working toward the PhD degree in software engineering with the School of Information Engineering, Yangzhou University, Yangzhou, China. His research interests include software security and deep learning. Some of his publications have been published in the top-tier conferences (e.g., ICSE, S&P, USENIX Security) and journals (e.g., *ACM Transactions on Software Engineering and Methodology*, *IEEE Transactions on Industrial Informatics*).

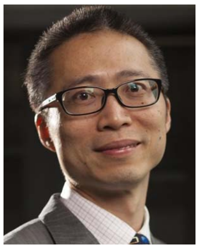
Xiaoxue Wu received the PhD degree from Northwestern Polytechnical University, Xi'an, China, in 2021. She is currently an associate professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. Her research interests include software testing and software security, etc.

Lili Bo received the PhD degree from the School of Computer Science and Technology, China University of Mining and Technology, in 2019. She is currently an associate professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. Her research interests include software testing and software security, etc.

Di Wu (Senior Member, IEEE) is a lecturer with the School of Mathematics, Physics, and Computing, University of Southern Queensland and a visiting fellow atwith the School of Computer Science, University of Technology Sydney. Prior to that, he was a researcher with the Australian Institute for Machine Learning (AIML) & School of Computer Science, University of Adelaide, Adelaide, Australia. Previous to this, he was an associate research fellow, Artificial Intelligence with Deakin Blockchain Innovation Lab, School of Information Technology, Deakin University, Melbourne, Australia, and worked as a postdoc fellow with the School of Computer Science, University of Technology Sydney (UTS), Sydney, Australia. He has more than 10 years of experience in research & development and academia. He has substantial industry experience in large project management, software development, and large system maintenance experience while working on various projects at China Telecom (Global 500), Shanghai. His research area focuses on applying AI on edge devices and AI applications. He has published more than 20 papers in refereed books, conferences, and journals. He has served as a special session chair for IJCNN. He also serves as a reviewer for many high-quality academic conferences and journals, such as CoRL, PR, TETCI, and so on.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.

Bin Li received the BS degree in computer software from Fudan University, Shanghai, China, in 1986, the MS and PhD degrees in computer application technology from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1993 and 2001, respectively. He is currently a professor with Yangzhou University, Yangzhou, China. He has authored and co-authored more than 100 journal and conference papers. His main research interests include knowledge graph, software data mining, and multiagent system.



Yang Xiang (Fellow, IEEE) received the PhD degree in computer science from Deakin University, Burwood, VIC, Australia, in 2007. He is currently a full professor and the dean of Digital Research, Swinburne University of Technology, Hawthorn, VIC, Australia. In the past 20 years, he has authored or coauthored more than 300 research papers in many international journals and conferences. His research focuses on cyber security, which covers network and system security, data analytics, distributed systems, and networking. He is the editor-in-chief of the Springer Briefs on Cyber Security Systems and Networks. He is an associate editor for *IEEE Transactions on Dependable and Secure Computing*, *IEEE Internet of Things Journal*, and *ACM Computing Surveys*. He is the Coordinator of the Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP), and the Chair of the Australia and New Zealand, IEEE Blockchain Technical Community.

Authorized licensed use limited to: Nanjing Univ of Post & Telecommunications. Downloaded on November 13,2025 at 08:01:26 UTC from IEEE Xplore. Restrictions apply.