Slice-Based Code Change Representation Learning

Fengyi Zhang, Bihuan Chen, Yufei Zhao, Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China

***Abstract*—Code changes are at the very core of software development and maintenance. Deep learning techniques have been used to build a model from a massive number of code changes to solve software engineering tasks, e.g., commit message generation and bug-fix commit identification. However, existing code change representation learning approaches represent code change as lexical tokens or syntactical AST (abstract syntax tree) paths, limiting the capability to learn semantics of code changes. Besides, they mostly do not consider noisy or tangled code change, hurting the accuracy of solved tasks. To address the above problems, we first propose a slice-based code change representation approach which considers data and control dependencies between changed code and unchanged code. Then, we propose a pre-trained sparse Transformer model, named CCS2VEC, to learn code change representations with three pre-training tasks. Our experiments by fine-tuning our pre-trained model on three downstream tasks have demonstrated the improvement of CCS2VEC over the state-of-the-art CC2VEC.**

***Index Terms*—code change, code slice, representation learning**

I. Introduction

The source code of software systems is continuously evolving by adding new features, fixing bugs, improving performance, or conducting refactoring, often in the form of commits. Therefore, code changes are at the very core of software development and maintenance. They usually need to be understood by developers when performing daily software development and maintenance tasks [44]. To ease the burden of developers, deep learning techniques have been used to build a model from a massive number of code changes. This model can be used to automate software engineering tasks, commit message generation [22, 28, 29], bug-fix commit identification [19], security patch identification [33, 61, 64], just-in-time defect prediction [17], to name but a few.

Code change representation learning plays a key role in these deep learning-based approaches. On one hand, general-purpose code change representation learning approaches have been proposed to solve multiple software engineering tasks [4, 18, 34, 38, 48]. While some of their models [4, 34] can be fine-tuned to solve different tasks, others [18, 38, 48] need to be retrained for different tasks. On the other hand, task-specific code change representation learning approaches have been widely explored, e.g., commit message generation [22] and just-in-time defect prediction [17]. They often lack the generality to solve other tasks.

One common problem of the two types of approaches is that they represent code change as a sequence of code tokens or AST (abstract syntax tree) paths, and hence have limited capability to learn semantics of code change with respect to control and data dependencies. However, these dependencies are important for understanding code change and solving tasks. For example, the control flow between the definition and a newly-added usage of a variable helps to determine whether the newly-added variable usage introduces a null pointer defect. Another common problem is that they mostly do not take into account noisy or tangled code change [16, 23], and hence impact the accuracy of solved tasks. For example, bug-fix code change and irrelevant refactoring code change are tangled in one commit, which may impact bug-fix commit identification models. The other problem is the lack of labeled data of code changes, especially for bug-related tasks (e.g., just-in-time defect prediction), although the data of code changes is actually massive from open-source projects.

To overcome these problems, we propose a slice-based code change representation approach which further takes into account the unchanged code that has control or data dependencies to the changed code. In particular, we conduct forward and backward slicing on the program dependence graph (PDG) of the code before (resp. after) the change according to the removed (resp. added) code. Then, we represent the code before (resp. after) the change as a sequence of slice, control flow paths, statements, and code tokens with hierarchical structure. In other words, the control flow paths are from the slice, the statements are from the control flow paths, and the code tokens are from the statements. Based on this representation, we propose a pre-trained sparse Transformer model CCS2VEC for code changes. Specifically, we follow the sparse Transformer [27] to incorporate the hierarchical structure of code changes, i.e., we use the hierarchical structure to decide the sparse pattern. We pre-train the model using three pre-training tasks to learn code change representations. Based on our pre-trained model, we fine-tune it to solve various downstream tasks. For a commit-level task, we adopt a Multiple Instance Learning pooling layer [20] before the task model to learn commit representations from multiple code changes in commits. In this way, noisy code changes in a commit can have a small contribution to the final commit representation.

We have conducted a set of experiments to evaluate the effectiveness of our pre-trained model, which is learned from 2,917K code changes from 234K commits from 109 open-source Java projects. We compare CCS2VEC with the state-of-the-art code change representation learning approach CC2VEC [18] on three downstream tasks. Our results have demonstrated that CCS2VEC outperforms CC2VEC by 25.2% in BLEU-4 score on the task of commit message generation, and 4.5% and 20.8% in F1-score respectively on the task of bug-fix commit identification and just-in-time defect prediction. Besides, we conduct an ablation study to reveal the contribution of the design decisions in our approach, as well as a sensitivity analysis to measure the sensitivity of our approach to its configurable parameters.

In summary, this work makes the following contributions.

- We proposed a slice-based code change representation approach, which considers the data and control dependencies between changed code and unchanged code.

- We proposed a pre-trained sparse Transformer model, named CCS2VEC, to learn code change representations.
- We learned the pre-trained model, and fine-tuned it on three downstream tasks to show its improvement over CC2VEC.

II. Preliminaries and Motivation

We first introduce preliminary knowledge related to our approach, then present a motivating example to illustrate our ideas.

A. Preliminaries

Vanilla Transformer. The vanilla Transformer [50] follows an encoder-decoder architecture using stacked self-attention and point-wise, fully connected layers for the encoder and decoder. The information fusing with self-attention can be viewed as message passing on a fully connected graph, where the input tokens are viewed as nodes and the attentions as edges. Such a perspective provides us with a chance to encode different graph structures by incorporating the inductive bias of self-attention.

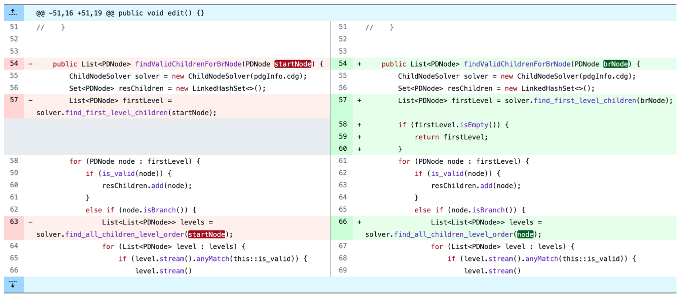
Sparse Transformer. In the default self-attention, each token attends to all other tokens in the input sequence. However, it is observed that in well-trained Transformers, the learned attention matrix tends to be sparse [8], i.e., most of the data points in the matrix are not attended over. This observation leads to a line of work on sparse Transformers that aim to reduce the computation complexity by incorporating structural bias to limit the number of query-key pairs that each query attends to [27]. Most of the existing sparse Transformers reduce the number of attending pairs through a pre-defined or random sparse pattern.

Multiple Instance Learning. Multiple Instance Learning (MIL) [20] refers to the task of predicting a label Y given a bag of instances $X = \{x_1, x_2, ..., x_K\}$, where the instances exhibit neither dependency nor ordering among each other. The process that maps representations of X to Y is referred to as MIL pooling. A typical approach to solve the task is the embedding-based approach, which first learns the representation of each instance, then uses MIL pooling to obtain a bag representation that is independent of the permutation of instances in the bag, and finally leverages a bag-level classifier to produce the final prediction. An attention mechanism is often adopted in MIL pooling to aid learning and provide weights among the instances, indicating the contribution of each instance to the final prediction.

B. A Motivating Example

We present an actual commit in Fig. 1 to illustrate the motivation of our approach. This commit contains three *hunks* starting at Line 54, 57 and 63, respectively. Each hunk is a span of consecutive lines of changed code, and we refer to a hunk as a code change. The first hunk replaces variable name `startNode` to `brNode`. The second hunk replaces variable `startNode` to `brNode`, and adds an empty check on variable `firstLevel`. The last hunk replaces variable name `startNode` to `node`.

A comprehensive semantic understanding is needed to capture the intention behind these code changes. At Line 57, both variables `startNode` and `brNode` are linked to the same definition at Line 54 by data flow, which suggests that the hunk is a variable renaming rather than a variable replacement. However, at Line 63, variables `startNode` and `node` are lined to different definitions at Line 54 and 61 by data flow, which suggests that the hunk is about variable replacement rather than variable renaming. Besides, there is no validation in the control flow path between the definition and usage of variable `firstLevel`, it suggests a potential risk and an empty check before the usage of `firstLevel` is needed. We can also observe that not all the hunks correspond to the commit message "bug fix: variable misuse". In fact, only the last hunk is relevant to the commit message, while the other two hunks are tangled code changes.



```
@@ -51,16 +51,19 @@ public void edit() {}
51    //    }
52
53
54 -      public List<PDNode> findValidChildrenForBrNode(PDNode startNode) {
55            ChildNodeSolver solver = new ChildNodeSolver(pdgInfo.cdg);
56            Set<PDNode> resChildren = new LinkedHashSet<>();
57 -          List<PDNode> firstLevel =
   solver.find_first_level_children(startNode);
58            for (PDNode node : firstLevel) {
59                if (is_valid(node)) {
60                    resChildren.add(node);
61                }
62                else if (node.isBranch()) {
63 -                  List<List<PDNode>> levels =
   solver.find_all_children_level_order(startNode);
64                    for (List<PDNode> level : levels) {
65                        if (level.stream().anyMatch(this::is_valid)) {
66                            level.stream()
```

```
51    //    }
52
53
54 +      public List<PDNode> findValidChildrenForBrNode(PDNode brNode) {
55            ChildNodeSolver solver = new ChildNodeSolver(pdgInfo.cdg);
56            Set<PDNode> resChildren = new LinkedHashSet<>();
57 +          List<PDNode> firstLevel = solver.find_first_level_children(brNode);

58 +          if (firstLevel.isEmpty()) {
59 +              return firstLevel;
60 +          }
61            for (PDNode node : firstLevel) {
62                if (is_valid(node)) {
63                    resChildren.add(node);
64                }
65                else if (node.isBranch()) {
66 +                  List<List<PDNode>> levels =
   solver.find_all_children_level_order(node);
67                    for (List<PDNode> level : levels) {
68                        if (level.stream().anyMatch(this::is_valid)) {
69                            level.stream()
```

Fig. 1: A Motivating Example of Code Changes

This example shows the obstacles of existing code change representation learning approaches (e.g., [17, 18, 22, 38]) in learning semantics under code changes. They use lexical code tokens or syntactical AST paths to represent code changes, and hence it is non-trivial for them to learn the control and data dependencies between changed code and unchanged code.

Motivated by these observations, we propose a pre-trained approach that leverages semantic structure of code changes to learn code change representations. Our approach is built upon (i) a slice-based code change representation to capture control and data dependencies between changed code and unchanged code (see Sec. III-B), (ii) a pre-trained sparse Transformer model to learn code change representations (see Sec. III-C), and (iii) a fine-tuning process that uses labeled datasets to solve downstream tasks while weighing the importance of multiple code changes in a commit for commit-level tasks (see Sec. III-D).

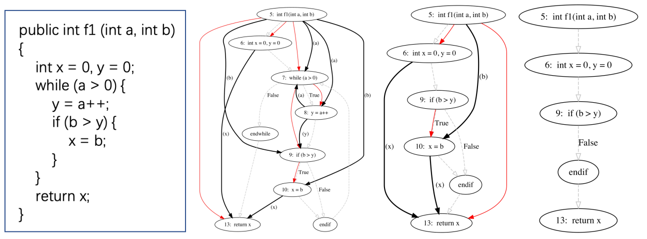
III. Methodology

We first present an overview of our approach, and then explain each step of our approach in detail.

A. Approach Overview

At a high level, our goal is to represent a code change in such a way that it can capture semantic information behind the code change, enable learning across massive historical code changes, and be used for different downstream tasks. To this end, we propose a pre-trained approach that leverages semantic structure of code changes to learn their representations. Specifically, our approach consists of the following three main steps.

- **Slice-Based Code Change Representation.** Given a commit, for each changed method, we build a program dependence graph (PDG) of the method before (resp. after) the change at the statement level without the need of a build environment. Then, for each hunk in each changed method, we conduct



```
public int f1 (int a, int b)
{
    int x = 0, y = 0;
    while (a > 0) {
        y = a++;
        if (b > y) {
            x = b;
        }
    }
    return x;
}
```

(a) Code (b) PDG (c) Slice (d) Path

Fig. 2: An Example of Code Change Slice and Representation

slicing on the PDG before (resp. after) the change according to the removed (resp. added) statements in the hunk. Finally, we represent the hunk before (resp. after) the change as a sequence of slice, control flow paths in the slice, statements in the control flow paths, and code tokens in the statements; i.e., we represent the hunk with hierarchical structure.

- **Sparse Transformer Model Pre-Training.** With our slice-based representation, we propose a pre-trained sparse Transformer model CCS2VEC to learn code change representations. In CCS2VEC, we use the hierarchical structure knowledge in our code change representation to decide the sparse pattern of the sparse Transformer, and use positional encoding to consider the ordering information in the sequence. We pre-train our model using three pre-training tasks.
- **Downstream Task Fine-Tuning.** We fine-tune our model to solve various downstream tasks. Specifically, for a commit-level task, we add a MIL pooling layer before the task model to learn commit representations from multiple hunks in commits. In this way, noisy hunks in a commit can have a small contribution to the final commit representation.

B. Slice-Based Code Change Representation

For a code change, our interest is "which part of the program is relevant to the changed part?". To reveal this semantic, we leverage data and control dependencies among statements inside a program to track the relevant part, namely a *code change slice*. We use two sub-steps, i.e., *PDG construction* and *data and control dependency tracking*, to produce such a slice. To better learn the semantics under the slice, we further take advantage of the hierarchical structure in the slice to represent a code change (i.e., the last sub-step *code change representation*).

Step 1: PDG Construction. For each changed method m in a commit, we construct a PDG $\mathcal{G}_{old}$ for the method before the change (denoted as m_{old}) and a PDG $\mathcal{G}_{new}$ for the method after the change (denoted as m_{new}). We define the PDG of a method as a 11-tuple $\mathcal{G} = \langle \mathcal{N}, n_s, \mathcal{N}_e, \mathcal{N}_c, \mathcal{T}, \mathcal{C}, \mathcal{DD}, \mathcal{CD}, \iota, \tau, \psi \rangle$. In particular, $\mathcal{N}$ denotes a set of nodes, and each node $n \in \mathcal{N}$ denotes a statement in the method source code. n_s denotes the entry node of $\mathcal{G}$, and we use method declaration statement as the entry node. For example, Fig. 2a shows the code of a method m_{new}, where `return x` is the added code; and Fig. 2b shows its PDG, where node 5 is the entry node. $\mathcal{N}_e$ denotes a set of exit nodes of $\mathcal{G}$. We regard return statement and throw statement as exit nodes, and set an exit node at the end of method execution if there is no return or throw statement. For example, node 13 in Fig. 2b is an exit node. $\mathcal{N}_c$ denotes a set of nodes that are changed in the commit. Specifically, in $\mathcal{G}_{old}$ (resp. $\mathcal{G}_{new}$), $\mathcal{N}_c$ corresponds to the statements that are removed (resp. added) in m_{old} (resp. m_{new}) in the commit. For example, node 13 is the added node. $\mathcal{T}$ denotes a set of source code tokens in the method. $\mathcal{C} : \mathcal{N} \times \mathcal{N}$ denotes a set of control flow edges, and each edge $c = n_1 \hookrightarrow n_2 \in \mathcal{C}$ denotes a control flow between n_1 and n_2. The dotted gray lines in Fig. 2b are control flow edges. $\mathcal{DD} : \mathcal{N} \times \mathcal{N}$ denotes a set of data dependency edges, and each edge $dd = n_1 \hookrightarrow n_2 \in \mathcal{DD}$ denotes a data dependency between n_1 and n_2. The black solid lines in Fig. 2b are data dependency edges. $\mathcal{CD} : \mathcal{N} \times \mathcal{N}$ denotes a set of control dependency edges, and each edge $cd = n_1 \hookrightarrow n_2 \in \mathcal{CD}$ denotes a control dependency, indicating that n_1 is a control structure statement (e.g., a `for` loop) and n_2 is a statement inside the control structure. The red solid lines in Fig. 2b are control dependency edges. ι denotes a function that maps each node $n \in \mathcal{N}$ to a sequence of source code tokens $\iota(n)$ in the corresponding statement. τ denotes a function that maps each node $n \in \mathcal{N}$ to a set of variables $\tau(n)$ that are defined in n. For example, node 8 in Fig. 2b has a definition of variable `y`. ψ denotes a function that maps each node $n \in \mathcal{N}$ to a set of variables $\psi(n)$ that are used in n. For example, node 8 in Fig. 2b has a usage of variable `a`.

We construct $\mathcal{G}_{old}$ and $\mathcal{G}_{new}$ by a lightweight tool PROGEX [14], which works at the source code level without the need of a build environment. We use the out-degree of nodes with respect to control flow edges to determine the exit node set $\mathcal{N}_e$, and use the diff in the commit to determine the changed node set $\mathcal{N}_c$.

Step 2: Data and Control Dependency Tracking. After constructing $\mathcal{G}_{old}$ for m_{old} and $\mathcal{G}_{new}$ for m_{new}, we represent the code changes in m in the commit as a set of hunks $\mathcal{H}$, and each hunk $h \in \mathcal{H}$ has a set of consecutive changed nodes (i.e., the removed (resp. added) nodes $\mathcal{N}_{rem}$ (resp. $\mathcal{N}_{add}$) belong to $\mathcal{N}_c$ of $\mathcal{G}_{old}$ (resp. $\mathcal{G}_{new}$). For each hunk h, we first track the nodes $\mathcal{N}_{old}^{data}$ (resp. $\mathcal{N}_{new}^{data}$) that have data dependency with the nodes in $\mathcal{N}_{rem}$ (resp. $\mathcal{N}_{add}$) on $\mathcal{G}_{old}$ (resp. $\mathcal{G}_{new}$). Specifically, for each removed (resp. added) node $n_i \in \mathcal{N}_{rem}$ (resp. $\mathcal{N}_{add}$), we conduct forward and backward reachability analysis on $\mathcal{G}_{old}$ (resp. $\mathcal{G}_{new}$) with respect to the data dependency edges in $\mathcal{DD}$, and add the reached nodes to $\mathcal{N}_{old}^{data}$ (resp. $\mathcal{N}_{new}^{data}$). Therefore, $\mathcal{N}_{old}^{data}$ (resp. $\mathcal{N}_{new}^{data}$) is a union of the reached nodes of all the removed (resp. added) nodes in $\mathcal{N}_{rem}$ (resp. $\mathcal{N}_{add}$). Here we set a threshold value K_{data} to limit the maximum depth during data dependency tracking. For example, for the added node 13 in Fig. 2b, we add nodes 5, 6 and 10 to $\mathcal{N}_{new}^{data}$.

Then, we track the nodes $\mathcal{N}_{old}^{ctrl}$ (resp. $\mathcal{N}_{new}^{ctrl}$) that have control dependency with any node in $\mathcal{N}_{old}^{data} \cup \mathcal{N}_{rem}$ (resp. $\mathcal{N}_{new}^{data} \cup \mathcal{N}_{add}$). Therefore, for each node $n \in \mathcal{N}_{old}^{data} \cup \mathcal{N}_{rem}$ (resp. $\mathcal{N}_{new}^{data} \cup \mathcal{N}_{add}$), we conduct backward reachability analysis on $\mathcal{G}_{old}$ (resp. $\mathcal{G}_{new}$) with respect to the control dependency edges in $\mathcal{CD}$. For each reached node n_{cur}, if there exists an edge $n_{cur} \hookrightarrow n \in \mathcal{CD}$, we add n_{cur} to $\mathcal{N}_{old}^{ctrl}$ (resp. $\mathcal{N}_{new}^{ctrl}$). It means that n_{cur} is the direct control structure that encapsulates n, and hence we always take it into account. Otherwise, we calculate the predecessor sibling nodes $\mathcal{N}_{pred}$ of n_{cur} (i.e., the nodes that are executed before n_{cur} and under the same residing control structure of n_{cur}). If there exists any node $n' \in N_{pred}$ such that

$\tau(n') \cap \psi(n) \neq \emptyset$, we add n_{cur} to $\mathcal{N}_{old}^{ctrl}$ (resp. $\mathcal{N}_{new}^{ctrl}$). It means that for a higher-order control structure, if the control structure has a predecessor sibling statement that has the definition of a variable used at n, we consider the control structure as having influence on n. Here we also set a threshold value K_{ctrl} to limit the maximum depth during tracking. For example, node 9 in Fig. 2b is a direct control structure of node 10, and hence node 9 is added to $\mathcal{N}_{new}^{ctrl}$. However, node 7 is a higher-order control structure of node 10, and its predecessor sibling node 6 does not define the variable `b` that is used in node 10, and thus node 7 is not added to $\mathcal{N}_{new}^{ctrl}$.

Finally, for each hunk h, we remove from m_{old} (resp. m_{new}) the statements that are not included in $\mathcal{N}_{old}^{data} \cup \mathcal{N}_{old}^{ctrl} \cup \mathcal{N}_{rem}$ (resp. $\mathcal{N}_{new}^{data} \cup \mathcal{N}_{new}^{ctrl} \cup \mathcal{N}_{add}$) (i.e., we remove the statements that are not relevant to the changed code in h) to get the method after slicing m'_{old} (resp. m'_{new}). We define m'_{old} (resp. m'_{new}) as a code change slice $\mathcal{S}_{old}$ (resp. $\mathcal{S}_{new}$) for the hunk h.

Step 3: Code Change Representation. Given a code change slice $\mathcal{S}_{old}$ (resp. $\mathcal{S}_{new}$), we represent it as a sequence $\mathcal{L}_{old}$ (resp. $\mathcal{L}_{new}$) $= (s, p_1, ..., p_x, n_1, ..., n_y, t_1, ..., t_z)$.

In particular, s denotes a PDG of the code change slice. We define s as a 8-tuple $\langle \mathcal{N}, n_s, \mathcal{N}_e, \mathcal{T}, \mathcal{C}, \mathcal{DD}, \mathcal{CD}, \iota \rangle$, where $\mathcal{N}$, n_s, $\mathcal{N}_e$, $\mathcal{T}$, $\mathcal{C}$, $\mathcal{DD}$, $\mathcal{CD}$ and ι share the same meaning to those in $\mathcal{G}$. Notice that $\mathcal{N} = \mathcal{N}_{old}^{data} \cup \mathcal{N}_{old}^{ctrl} \cup \mathcal{N}_{rem}$ (resp. $\mathcal{N}_{new}^{data} \cup \mathcal{N}_{new}^{ctrl} \cup \mathcal{N}_{add}$) in s_{old} (resp. s_{new}). Similar to $\mathcal{G}$, we leverage PROGEX [14] to generate s. For example, Fig. 2c shows the PDG of the code change slice generated from Fig. 2a.

Further, $p_1...p_x$ denotes all possible control flow paths from n_s to a node in $\mathcal{N}_e$ in s. To produce finite control flow paths, we restrict that no control flow path contains a cycle. For example, there are two possible control flow paths in Fig. 2c, and one of them is shown in Fig 2d. $n_1...n_y$ denotes nodes in $\mathcal{N}$ ordered by the line number of the corresponding statements. $t_1...t_z$ denotes source code tokens in all nodes (i.e., statements).

We can observe that we represent a code change slice in such a way that it contains multiple types with rich hierarchical structure relations for the ease of semantic understanding. Specifically, we define the following eight types of relation sets.

- $\mathcal{R}_1$ denotes the relations between the graph of a code change slice s and the control flow paths $p_1...p_x$. Each relation $s \hookrightarrow p_i \in R_1$ indicates that a control flow path p_i is contained in s.
- $\mathcal{R}_2$ denotes the relations between $p_1...p_x$ and the statement nodes $n_1...n_y$. Each relation $p_i \hookrightarrow n_j \in R_2$ indicates that a statement node n_j is contained in a control flow path p_i.
- $\mathcal{R}_3$ denotes the relations between any two changed nodes in $\mathcal{N}_{rem}$ (resp. $\mathcal{N}_{add}$). Each relation $n_{ch}^i \hookrightarrow n_{ch}^j$ indicates a weight between two changed nodes n_{ch}^i and n_{ch}^j.
- $\mathcal{R}_4$ denotes the relations between $\mathcal{N}_{rem}$ (resp. $\mathcal{N}_{add}$) and $\mathcal{N}_{old}^{data}$ (resp. $\mathcal{N}_{new}^{data}$). Each relation $n_{ch} \hookrightarrow n_{data}$ indicates a data dependency between a changed node n_{ch} and n_{data}.
- $\mathcal{R}_5$ denotes the relations between two nodes in $\mathcal{N}_{old}^{data}$ (resp. $\mathcal{N}_{new}^{data}$). Each relation $n_{data}^i \hookrightarrow n_{data}^j$ indicates a data dependency between two nodes n_{data}^i and n_{data}^j.
- $\mathcal{R}_6$ denotes the relations between $\mathcal{N}_{rem} \cup \mathcal{N}_{old}^{data}$ (resp. $\mathcal{N}_{add} \cup \mathcal{N}_{new}^{data}$) and $\mathcal{N}_{old}^{ctrl}$ (resp. $\mathcal{N}_{new}^{ctrl}$). Each relation $n \hookrightarrow n_{ctrl}$ indicates a control dependency between n and n_{ctrl}.

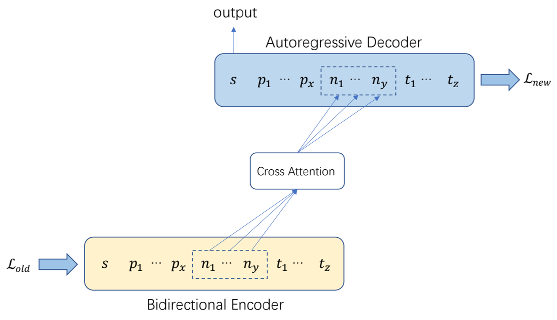


Fig. 3: Model Architecture of CCS2VEC

- $\mathcal{R}_7$ denotes the relations between two nodes in $\mathcal{N}_{old}^{ctrl}$ (resp. $\mathcal{N}_{new}^{ctrl}$). Each relation $n_{ctrl}^i \hookrightarrow n_{ctrl}^j$ indicates a control dependency between two nodes n_{ctrl}^i and n_{ctrl}^j.
- $\mathcal{R}_8$ denotes the relations between $n_1...n_y$ and $t_1...t_z$. Each relation $n_i \hookrightarrow t_j$ indicates that the statement node n_i contains a source code token t_j.

Therefore, $\mathcal{L}$ can be considered as a graph $\mathcal{B}$ whose nodes are all the input tokens in $\mathcal{L}$ and whose edges are $\mathcal{R}_1 \cup \mathcal{R}_2 \cup ... \cup \mathcal{R}_8$. The nodes and edges in $\mathcal{B}$ have various types.

C. Sparse Transformer Model Pre-Training

With our slice-based code change representation, i.e., $\mathcal{L}_{old}$ and $\mathcal{L}_{new}$, for each hunk, we develop a pre-trained sparse Transformer model CCS2VEC to learn a representation for each hunk.

Model Architecture. Fig. 3 shows the model architecture of CCS2VEC, which adopts an encoder-decoder Transformer as the overall architecture. The encoder is a bidirectional model, and the decoder is an autoregressive model. The sequence $\mathcal{L}_{old}$ is encoded with the bidirectional encoder, and then the likelihood of the sequence $\mathcal{L}_{new}$ is computed with the autoregressive decoder. Each layer of the decoder performs a cross-attention over the final hidden layer of the encoder. Overall, our approach follows the design of BART architecture [24].

Specifically, the encoder and decoder share a similar Transformer module which takes a sequence $\mathcal{L}$ as input. It first converts the sequence into an initial input embedding H^0, where the embedding of each token in the sequence is calculated by adding up the initial token embedding and the positional embedding. Then, it applies N Transformer layers on H^0, where each Transformer is identical in structure that applies a multi-head graph self-attention layer followed by a feed forward layer. The output of the n-th Transformer layer H^n is obtained by Eq. 1, where $norm(\cdot)$ denotes a layer normalization, $GSA(\cdot)$ denotes the multi-head graph self-attention, and $FFN(\cdot)$ denotes a two-layer feed forward network.

$$\begin{aligned} G^n &= norm(GSA(H^{n-1}) + H^{n-1}) \\ H^n &= norm(FFN(G^n) + G^n) \end{aligned} \tag{1}$$

For the n-th Transformer layer, $GSA(\cdot)$ is obtained by Eq. 2, where the previous Transformer layer's output H^{n-1} is linearly projected into the queries matrix by learnable parameter matrix W_i^Q. Another output of the previous Transformer layer A^{n-1} is linearly projected into the keys and values matrixes by learnable parameter matrixes W_i^K and W_i^V respectively. A^{n-1} denotes

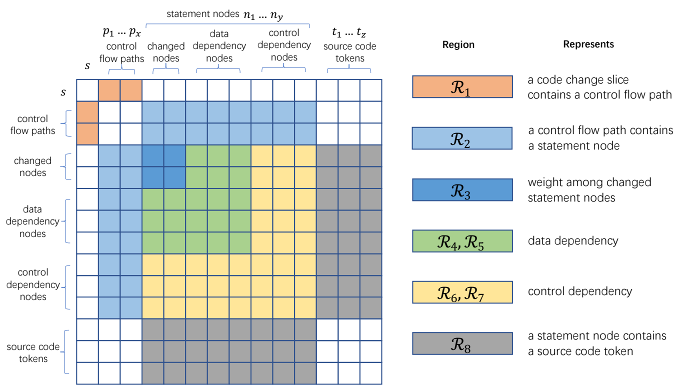

statement nodes $n_1 \dots n_y$
$p_1 \dots p_x$ control flow paths
changed nodes
data dependency nodes
control dependency nodes
$t_1 \dots t_z$ source code tokens
s
control flow paths
changed nodes
data dependency nodes
control dependency nodes
source code tokens
Region
Represents
$\mathcal{R}_1$ a code change slice contains a control flow path
$\mathcal{R}_2$ a control flow path contains a statement node
$\mathcal{R}_3$ weight among changed statement nodes
$\mathcal{R}_4, \mathcal{R}_5$ data dependency
$\mathcal{R}_6, \mathcal{R}_7$ control dependency
$\mathcal{R}_8$ a statement node contains a source code token

Fig. 4: The Mask Matrix

the neighborhood representations in the previous Transformer layer. For each node u in $\mathcal{B}$, we calculate A_u^{n-1} by concatenating the hidden state of u's neighborhoods $\mathcal{A}(u)$ in $\mathcal{B}$ (i.e., $\mathcal{A}(u)$ contains the nodes in $\mathcal{B}$ which u has a relation to). M denotes a mask matrix and PE denotes the relative positional encoding of all the nodes in $\mathcal{B}$, which are introduced in the following sections. $i \in [1, a]$ and a denotes the number of attention heads; and W_n^O denotes a learnable parameter matrix.

$$
\begin{aligned}
A^{n-1} &= [A_1^{n-1}; ...; A_{|L|}^{n-1}] \\
A_u^{n-1} &= concat(\{h_v^{n-1} \mid v \in \mathcal{A}(u)\}) \\
Q_i = H^{n-1}W_i^Q, K_i &= A^{n-1}W_i^K, V_i = A^{n-1}W_i^V \\
head_i &= softmax(\frac{Q_i(K_i + PE)^T}{\sqrt{d_k}} + M)V_i \\
GSA(\mathcal{B}, H^{n-1}) &= [head_1, ..., head_a]W_n^O
\end{aligned}
\tag{2}
$$

Finally, we use the hidden state of s in $\mathcal{L}_{new}$ at the final decoder layer as the representation of the hunk. Here we choose to use s in $\mathcal{L}_{new}$ rather than $\mathcal{L}_{old}$ as the representation because it is updated by $\mathcal{L}_{old}$ in the pre-training tasks.

Masked Self-Attention. As introduced in Sec. III-B, $\mathcal{L}$ contains rich hierarchical structure relations. We aim to incorporate such relations in $\mathcal{L}$ into the Transformer layer through the mask matrix M, resulting in a sparse Transformer. Specifically, we define the mask matrix M (i.e., the sparse pattern) by allowing each token in $\mathcal{L}$ to only attend over the tokens which it has a relation to. Specifically, we define M by Eq. 3,

$$
M_{i,j} = \begin{cases} 0 & \exists\ e_{i,j} \in \mathcal{R}_1 \cup \mathcal{R}_2 \cup ... \cup \mathcal{R}_8 \\ -\infty & otherwise \end{cases}
\tag{3}
$$

where $e_{i,j}$ denotes a relation between the i-th token and j-th token in $\mathcal{L}$. If $e_{i,j}$ belongs to $\mathcal{R}_1$, $\mathcal{R}_2$, ..., or $\mathcal{R}_8$, $M_{i,j}$ is set to 0, which means that the i-th token in $\mathcal{L}$ is allowed to attend over the j-th token. Otherwise, $M_{i,j}$ is set to $-\infty$, which means the i-th token in $\mathcal{L}$ is not allowed to attend over the j-th token. As visually presented in Fig. 4, each defined relation locates to a region in M, and the blank regions indicate no relation between tokens, and thus the attention is forbidden. Self-attention of a token over its own is not shown for simplicity in Fig. 4.

Positional Encoding. We further consider the orders of the tokens in $\mathcal{L}$. As $\mathcal{L}$ contains multiple relation types, it is not feasible to use the vanilla absolute positional encoding. As suggested by Shaw et al. [43], introducing the relative distances between tokens helps capture their relative orders. We draw an analogy on the underlying graph $\mathcal{B}$ of $\mathcal{L}$, and define multiple latent distance representations for different relation types.

Specifically, for each token v in $\mathcal{A}(u)$, we consider the relative positional difference in $\mathcal{B}$ between u and v, and assign multiple latent representations $r_{u,v}$ on such difference according to the relation type, as formulated in Eq. 4,

$$
r_{u,v} = \begin{cases} r^{ch} & \exists\ e_{u,v} \in R_3 \\ r^{data} & \exists\ e_{u,v} \in R_4 \cup R_5 \\ r^{ctrl} & \exists\ e_{u,v} \in R_6 \cup R_7 \\ r^{src} & \exists\ e_{u,v} \in R_8 \end{cases}
\tag{4}
$$

where $e_{u,v}$ denotes a relation between token u and token v, and r^{ch}, r^{data}, r^{ctrl} and r^{src} are trainable parameters. Here we do not assign $r_{u,v}$ for s and $p_1, ..., p_x$ in $\mathcal{L}$ because there is only one code change slice in $\mathcal{L}$ and the control flow paths do not embody any order information. Finally, we define the relative positional encoding PE by Eq. 5,

$$
\begin{aligned}
PE^u &= concat(\{r_{u,v} \mid v \in A(u)\}) \\
PE &= [PE^1; PE^2; ...; PE^{|L|}]
\end{aligned}
\tag{5}
$$

where $concat(\cdot)$ denotes a concatenating operation.

Pre-Training Tasks. We construct a sparse Transformer that incorporates different node types and edge types in $\mathcal{B}$ into self-attentions. To capture semantics of such a structure in $\mathcal{B}$, we use three pre-training tasks, i.e., *node reconstruction*, *edge reconstruction* and *code change translation*. Before explaining how to update learnable parameters by these pre-training tasks, we introduce the embedding initialization of each node in $\mathcal{B}$.

Node Embedding Initialization. Following BP-Transformer [58], we divide the nodes in $\mathcal{B}$ of $\mathcal{L}$ into two disjoint sets, token nodes and span nodes. Token nodes correspond to source code tokens $t_1...t_z$, and span nodes correspond to the PDG of a code change slice s, control flow paths $p_1...p_x$, and statement nodes $n_1...n_y$. The representations of span nodes are initialized with all zeros, and the representations of token nodes are initialized with their corresponding word embeddings.

Node Reconstruction Task. This task aims to update the representation of each node in $\mathcal{B}$ of the sequence $\mathcal{L}$, and is analogous to the masked language modeling task in BERT [9]. The difference is that we regard the sequence $\mathcal{L}$ as its underlying graph $\mathcal{B}$ and a node in $\mathcal{B}$ only accepts messages passed from its neighborhood nodes in $\mathcal{B}$. In this task, each region in $\mathcal{R}_1 \cup \mathcal{R}_2 \cup ... \cup \mathcal{R}_8$ serves as an bridge of message passing between nodes such that a node is only updated by nodes in specified regions. We randomly sample 15% of the nodes in $\mathcal{B}$, replace them with the $[MASK]$ token 80% of the time, replace them with a random token 10% of the time, and leave them unchanged 10% of the time. The objective of this task is a cross entropy loss between the predicted tokens and the original masked tokens.

Edge Reconstruction Task. This task aims to incorporate information from different edge types in the underlying graph $\mathcal{B}$ of $\mathcal{L}$. While we define eight types of relations in $\mathcal{B}$, we only use a subset of them, i.e., $\mathcal{R}_4$ and $\mathcal{R}_5$ that correspond to data dependency, and $\mathcal{R}_6$ and $\mathcal{R}_7$ that correspond to control dependency. We believe data and control dependency are of the most importance in understanding code change semantics, and hence they need specialized pre-training tasks. We first sample 20% of the edges from $\mathcal{R}_4$ and $\mathcal{R}_5$ for data dependency prediction and from $\mathcal{R}_6$ and $\mathcal{R}_7$ for control dependency prediction, then mask the sampled edges by adding a negative infinity value in the mask matrix, and predict the original edges. Specifically, we follow GraphCodeBERT [15] to define the objective of this task, as formulated by Eq. 6,

$$\begin{aligned} loss = -\sum_{e_{i,j} \in E_C} [\sigma(e_{i,j} \in E_{mask}) \log p_{e_{i,j}} + \\ (1 - \sigma(e_{i,j} \in E_{mask}) \log(1 - p_{e_{i,j}}))] \end{aligned} \tag{6}$$

where $e_{i,j}$ is an edge from the i-th token to the j-th token in $\mathcal{L}$; E_{mask} is the masked edges; E_C is a set of candidate edges for edge prediction, and we set E_C as edges in $\mathcal{R}_4$ and $\mathcal{R}_5$ for data dependency prediction and set E_C as edges in $\mathcal{R}_6$ and $\mathcal{R}_7$ for control dependency prediction; $\sigma(e)$ is 1 if $e \in E_{mask}$, otherwise $\sigma(e)$ is 0; the probability $p_{e_{i,j}}$ of existing an edge from the i-th token to the j-th token in $\mathcal{L}$ is calculated by Eq. 7,

$$p_{e_{i,j}} = sigm(h_i) \cdot sigm(h_j) \tag{7}$$

where $sigm(\cdot)$ is a sigmoid non-linearity, and h_i and h_j are representations of the i-th and j-th token.

Code Change Translation Task. This task aims to capture semantics of the code change from $\mathcal{L}_{old}$ to $\mathcal{L}_{new}$, and we use a translation task to achieve this goal. As illustrated in Fig. 3, the encoder and decoder are connected by a cross attention, and we restrict the cross attention to only the changed statement nodes such that the changed statement nodes in $\mathcal{L}_{new}$ attend over the changed statement nodes in $\mathcal{L}_{old}$. The motivation is to encourage our model to infer how statement nodes change in a hunk. The training objective is the cross entropy loss between the decoder's output and the changed statement nodes in $\mathcal{L}_{new}$.

D. Downstream Task Fine-Tuning

CCS2VEC is a pre-trained model for each hunk in a commit. However, downstream tasks are often conducted at the level of a commit which can include multiple hunks. To fill this gap, we add an extra fusion layer in the fune-tuning process to learn the representation of commit by considering the importance of each hunk to a given downstream task.

Specifically, we formulate the fusion of hunks in a commit as a Multiple Instance Learning (MIL) [20] problem which aims to learn a bag representation given a bag of independent instances. In our case, we treat each hunk as an instance. To learn the representation of a commit, we adopt a MIL pooling layer, which is formulated by Eq. 8,

$$\begin{aligned} Z = \sum_{k=1}^{K} \alpha_k h_k \\ \alpha_k = \frac{exp\{w^\top tanh(Vh_k^\top) \odot sigm(Uh_k^\top)\}}{\sum_{j=1}^{K} exp\{w^\top tanh(Vh_j^\top) \odot sigm(Uh_j^\top)\}} \end{aligned} \tag{8}$$

where h_k denotes the representation of the k-th hunk after pre-training; w, V and U are learnable parameters; $\odot$ is an element-wise multiplication and $tanh(\cdot)$ and $sigm(\cdot)$ are non-linearities.

The MIL pooling can be seen as a special version of attention mechanism between the final output and input instances. Thus, α_k weights the contribution of k-th hunk in representing Z. We choose to put the MIL pooling layer in fine-tuning rather than in pre-training because an attention weight matrix is dependent to a specific task, e.g., the k-th hunk might be of high importance in generating commit messages but of low importance in predicting defects. The final output Z is then fed into different downstream models for specific tasks.

IV. Evaluation

We have implemented our approach in 2.7K lines of Python code and 8.6K lines of Java code, using PyTorch for deep learning and PROGEX for PDG construction. We have released the source code of our approach at https://ccs2vec.github.io/ with the dataset used in our evaluation.

A. Evaluation Setup

Data Collection and Model Pre-Training. To pre-train our model, we first collect commits from high-quality open-source Java projects. To this end, we collect open-source Java projects by the following criteria to ensure their quality: (1) the projects must have more than 500 stars and forks; (2) the projects must have more than 1,000 commits; and (3) the projects must have more than 50 bug-fix commits. Here we use the keyword-based approach [39] to identify bug-fix commits. Finally, we collect 109 high-quality Java projects which have 234K commits and 38K bug-fix commits. From these 234K commits, we construct a data set of 2,917K hunks, and split this data set into training, validation and testing sets by 7:1:2 for pre-training.

Research Questions. To evaluate the effectiveness of our pre-trained model, we fine-tune the model on three downstream tasks, i.e., commit message generation, bug-fix commit identification and just-in-time defect prediction, which are commonly used in literature. Then, we compare the results of these tasks with the state-of-the-art code change representation learning approaches. Specifically, we design our evaluation to answer the following research questions.

- **RQ1:** What is the effectiveness of CCS2VEC in downstream tasks, compared with the state-of-the-art approaches?
- **RQ2:** What is the contribution of each component in CCS2VEC to the achieved effectiveness?
- **RQ3:** What is the sensitivity of CCS2VEC's effectiveness to its configurable parameters?

State-of-the-Art Approaches. We select CC2VEC [18] as the state-of-the-art approach of general-purpose code change representation learning. This approach has been empirically demonstrated to outperform the existing work on all three tasks. However, it is not a pre-trained model, and thus needs to be retrained with specific tasks. We follow the same experimental settings in CC2VEC on the three tasks. Moreover, we select the recent state-of-the-art approaches of task-specific code change representation learning, i.e., FIRA [11] for the task of commit message

TABLE I: Results on Commit Message Generation

Approach	Original Data Set	Cleaned Data Set
CC2VEC	32.2	16.2
FIRA	37.6	20.5
CCS2VEC-SIM	40.3	24.7
CCS2VEC-TRAN	42.4	26.2

generation, and JITLINE [36] and LAPREDICT [60] for the task of just-in-time defect prediction.

B. Task 1: Commit Message Generation (RQ1)

Problem Formulation. Commit messages are important for software maintenance and program understanding. However, as reported by Dyer et al. [12], developers are not inclined to write commit messages that are standard and of high quality, and 14% of commit messages in projects on SourceForge were empty. Therefore, the task of automated commit message generation is needed, which aims to generate a message that summarizes the code changes in a given commit.

State-of-the-Art Approaches. We compare CCS2VEC with two state-of-the-art approaches, i.e., CC2VEC [18] and FIRA [11]. We further select FIRA because it is the most recent work in this topic and achieves the best performance. CC2VEC uses the learned representations to find a most relevant commit in the training set, and reuses its commit message as prediction. FIRA treats this task as a sequence-to-sequence translation problem.

Our Approach. To use CCS2VEC in this task, we first calculate the commit representation Z by the process described in Sec. III-D. Then, we use Z following the setting of each comparison approach for a fair comparison. In particular, to compare with CC2VEC, we use Z to find a most similar commit in training set by calculating cosine similarity, and reuse the found commit message as prediction. We name this version of our approach as CCS2VEC-SIM. To compare with FIRA, we feed Z as the final output of an encoder to a newly added decoder. The newly added decoder is identical to FIRA's, which is in charge of generating new commit message tokens. We name this version of our approach as CCS2VEC-TRAN.

Experimental Settings. To fine-tune our model as well as to train CC2VEC and FIRA, we extract commit messages from the 234K commits collected in Sec. IV-A. Specifically, we follow the same pre-processing approach in CC2VEC to remove merge and rollback commits, and commits that are too long. After the filtering, we collect an original data set of 175K commits and their messages. Furthermore, we adopt the method proposed by Liu et al. [30] to remove bot and trivial commit messages. After this filtering, we collect a cleaned data set of 87K commits and their messages. We use both the original and cleaned data sets for our evaluation, and split them into training, validation and testing sets by 7:1:2. Besides, we adopt BLEU-4 score to evaluate the quality of generated commit messages because BLEU-4 score has been widely used in previous work.

Results. We report the performance of CCS2VEC and state-of-the-art methods in Table I. CCS2VEC significantly outperforms the compared approaches in both settings. On the original data set, CCS2VEC-SIM outperforms CC2VEC by 25.2% in BLEU-4 score when it uses the same generator with CC2VEC. CCS2VEC-TRAN outperforms FIRA by 12.8% when it uses the same generator with FIRA. On the cleaned data set, CCS2VEC achieves a performance improvement of 52.5% comparing with CC2VEC and 27.8% comparing with FIRA, which are larger than on the original data set. These results indicate that CCS2VEC produces more semantic-aware representations than prior approaches.

CCS2VEC outperforms the state-of-the-art general-purpose code change representation learning approach CC2VEC by at least 25.2% in BLEU-4 score on the task of commit message generation, while outperforming the state-of-the-art task-specific approach FIRA by at least 12.8%.

C. Task 2: Bug-Fix Commit Identification (RQ1)

Problem Formulation. Bug-fix commits provides the base for various bug-related software maintenance tasks, e.g., bug-fix backporting from the latest version to older versions, bug prediction, and bug repairing. However, it is often hard to distinguish bug-fix commits from other commits such as feature addition or refactoring commits. Therefore, the task of automated bug-fix commit identification is needed. We regard this task as a binary classification problem, where each commit is labeled as a bug-fix commit or not based on the code changes in the commit.

State-of-the-Art Approaches. We compare our approach with CC2VEC [18], which first learns a meaningful representation, and then integrates the learned representations with two state-of-the-art task-specific approaches, i.e., PATCHNET [19] and LPU-SVM [47]. To use PATCHNET with CC2VEC, two additional embeddings are extracted from the commit message and code changes by PATCHNET, the vectors are concatenated with the embedding extracted by CC2VEC to form a new embedding, and the new embedding is fed into PATCHNET's classification module to predict whether a given commit is a bug-fix commit. To use LPU-SVM with CC2VEC, we pass the vectors produced by CC2VEC into the SVM classfier in LPU-SVM as features. Notice that the above setting is the same to the one in CC2VEC. As CC2VEC has already outperformed these two task-specific approaches, we do not compare our approach with them.

Our Approach. To use CCS2VEC in this task, we first calculate the commit representation Z by the process in Sec. III-D, and then feed Z to a commit classifier. To fairly compare with prior works, we use the same classfiers with CC2VEC, namely the PATCHNET and the LPU-SVM. Notice that our approach does not need extra embedding on commit messages.

Experimental Settings. To fine-tune our model as well as to train CC2VEC, we use the 38K bug-fix commits collected in Sec. IV-A as the positive data set, and randomly select the same number of commits from the remaining commits (i.e., the 234K commits − the 38K commits) as the non-bug-fix commits (i.e., the negative data set). We also split the data set into training, validation and testing sets by 7:1:2. Besides, following prior works, we use the following four commonly-used metrics to measure the effectiveness of bug-fix commit classification.

- *Accuracy*: the ratio between the number of correct predictions and the number of total predictions.

TABLE II: Results on Bug-Fix Commit Identification

Approach	Acc.	Prec.	Rec.	F1
CC2VEC + PATCHNET	78.2	77.9	72.1	74.9
CC2VEC + LPU-SVM	65.1	64.5	62.6	63.4
CCS2VEC + PATCHNET	76.5	78.4	78.2	78.3
CCS2VEC + LPU-SVM	69.1	71.9	70.2	71.5

- *Precision*: the ratio between the number of correct predictions on bug-fix commits and the total number of bug-fix commit predictions.
- *Recall*: the ratio between the number of correct predictions on bug-fix commits and the total number of bug-fix commits.
- *F1-score*: the harmonic mean between precision and recall.

Results. We show the performance of CCS2VEC and CC2VEC with two classifiers PATCHNET and LPU-SVM in Table II. We can see that when using PATCHNET as the classifier, CCS2VEC outperforms CC2VEC by help improving bug-fix commit identification by 0.6%, 8.4% and 4.5% in terms of precision, recall and F1-score, but suffers a slight decrease by 2.1% in accuracy. When using LPU-SVM as the classifier, CCS2VEC outperforms CC2VEC by 6.1%, 11.5%, 12.1% and 12.8% in terms of accuracy, precision, recall and F1-score. These results demonstrate that CCS2VEC is more capable of learning and understanding commit-level semantics than prior approaches.

CCS2VEC outperforms the state-of-the-art general-purpose code change representation learning approach CC2VEC by improving the state-of-the art bug-fix commit identification approaches by at least 4.5% in F1-score.

D. Task 3: Just-in-Time Defect Prediction (RQ1)

Problem Formulation. Just-in-time defect (JIT) defect prediction aims to identify commits that may potentially introduce a bug when the commits are submitted. In this way, developers can be notified about the potential bug in their commits at the very first place, and hence the overall testing and debugging effort can be reduced. Therefore, the task of automated JIT defect prediction is need. We formulate this task as a binary classification problem where each commit is labeled as bug-introducing or not given the code changes in the commit.

State-of-the-art Approaches. We compare our approach with three state-of-the-art approaches, i.e., CC2VEC [18], JITLINE [36] and LAPREDICT [60]. We further select JITLINE and LAPREDICT because they are the most recent work in this topic and achieve the best performance. Specifically, CC2VEC first learns a distributed representation of code changes, and then integrates the learned representation with an existing JIT defect prediction approach DEEPJIT [17]. The embedding extracted from CC2VEC is concatenated with two additional embeddings extracted from commit message and code changes by DEEPJIT, and the overall embedding is fed into DEEPJIT's classification layer to make the prediction. JITLINE uses the frequency of code tokens as code change feature, and LAPREDICT simply uses the number of added lines as code change feature. While being simple, they achieve competitive performance.

Our Approach. To use CCS2VEC in this task, we first calculate the commit representation Z by the process in Sec. III-D, and then feed Z to a commit classifier. For a fair comparison, we leverage the same classifier used in CC2VEC and JITLINE, namely the DEEPJIT classifier.

TABLE III: Results on Just-in-Time Defect Prediction

Approach	Acc.	Prec.	Rec.	F1
CC2VEC	60.5	66.8	58.9	62.6
JITLINE	60.3	62.9	55.6	59.0
LAPREDICT	66.4	72.7	68.5	70.5
CCS2VEC	78.2	79.5	72.1	75.6

Experimental Settings. To fine-tune our model as well as to train CC2VEC, JITLINE and LAPREDICT, we use the data set of defective commits from Zeng et al.'s work [60]. Specifically, we use the defective commits from the three Java projects (i.e., JDT, Platform and Gerrit) in their data set as our fine-tuning data set. As a result, we collect a total of 23K defective commits. We further randomly select the same number of commits from the remaining commits in the three Java projects as non-defective commits. We also split the data set into training, validation and testing sets by 7:1:2. Besides, similar to the previous task, we use accuracy, precision, recall and F1-score to measure the effectiveness of JIT defect prediction.

Results. We give the performance of CCS2VEC and state-of-the-art approaches in Table III. We can observe that CCS2VEC outperforms all the state-of-the-art approaches. In particular, CCS2VEC outperforms CC2VEC by 29.3%, 19.0%, 22.4% and 20.8% in terms of accuracy, precision, recall and F1-score. In addition, CCS2VEC outperforms JITLINE by 29.7%, 26.4%, 29.7% and 28.1% in terms of accuracy, precision, recall and F1-score, while outperforming LAPREDICT by 17.8%, 9.4%, 5.3% and 7.2%. These results show that CCS2VEC is effective in learning semantics under code changes.

CCS2VEC outperforms the state-of-the-art general-purpose code change representation learning approach CC2VEC by 20.8% in F1-score on the task of JIT defect prediction, while outperforming the state-of-the-art task-specific approaches JITLINE and LAPREDICT by at least 7.2%.

E. Ablation Study (RQ2)

We conduct an ablation study to understand the contribution of each step in constructing a code change slice to the achieved effectiveness, i.e., data dependency, control dependency and control flow path. We intentionally focus on these three aspects because we believe they play the key role in capturing semantics of code changes. Since these steps in constructing a code change slice have a sequential dependency order, i.e., one step relies on the result of its previous step, an ablation study is hard to conduct in the slice construction phase. Instead, we choose to set the values in the corresponding region in the mask matrix M in CCS2VEC to $-\infty$ to achieve the ablations of data dependency, control dependency and control flow path. Moreover, we also conduct an ablation study to show the contribution of the MIL pooling layer in fine-tuning because we adopt it to solve the noisy code change problem in downstream tasks.

For the first commit message generation task, we conduct our ablation study on the cleaned data set with the CCS2VEC-TRAN

TABLE IV: Results of Ablation Study

Approach	Task 1 (BLEU-4)		Task 2 (F1)		Task 3 (F1)	
	Abl.	Drop	Abl.	Drop	Abl.	Drop
w/o data	16.7	36.3%	55.1	29.6%	66.1	12.6%
w/o control	20.1	23.3%	64.9	17.1%	63.2	16.4%
w/o path	21.1	19.5%	58.5	25.3%	69.5	8.1%
w/o MIL	16.5	37.0%	57.6	26.4%	63.0	16.7%
CCS2VEC	26.2	–	78.3	–	75.6	–

approach. For the second bug-fix commit identification task, we conduct our ablation study with the CCS2VEC + PATCHNET approach, and present the results on F1-score. For the third JIT defect prediction task, we also report the results on F1-score. The results of our ablation study are shown in Table IV, where the column *Abl.* is the performance of the reduced version of our approach, and *Drop* is the performance degradation.

Removing Data Dependency. To remove data dependencies, we set the green region in M (see Fig. 4) to $-\infty$. We can see that after removing data dependencies, the performance of three tasks is dropped by 36.3%, 29.6% and 12.6%.

Removing Control Dependency. To remove control dependencies, we set the yellow region in M (see Fig. 4) to $-\infty$. We can observe that after removing control dependency, the performance of three tasks is dropped by 23.3%, 17.1% and 16.4%.

Removing Control Flow Path. To remove control flow paths, we set the orange and pale blue regions in M (see Fig. 4) to $-\infty$. This ablation leaves the slice node s an orphan node. To avoid it, we allow the slice node s to attend over all statement nodes. We find that after removing control flow paths, the performance of three tasks is dropped by 19.5%, 25.3% and 8.1%.

Removing MIL Pooling Layer. The MIL pooling layer aims to learn the commit representation Z by weighing the importance of hunks in the commit. To evaluate its effectiveness, we remove the MIL pooling layer in our model, and simply calculate Z by averaging the representations of the hunks in the commit. After removing the MIL pooling layer, the performance of three tasks is dropped by 37.0%, 26.4% and 16.7%.

Overall, data dependency, control dependency, and control flow path have a significant contribution to capture the semantics of code changes. MIP pooling layer is effective in solving the noisy code change problem.

F. Sensitivity Analysis (RQ3)

Our approach relies on two configurable parameters, K_{data} and K_{ctrl} in our slice-based code change representation approach (see Sec. III-B). Specifically, to determine the depth of data dependency tracking K_{data}, we set $K_{data} = \frac{|N|}{a} + 1$, where $|N|$ is the number of PDG nodes and a is a manually-set parameter. The definition indicates that K_{data} should be proportional to the length of the changed method, i.e., the longer the method is, the deeper we should track its data dependency. To determine the depth of control dependency tracking K_{ctrl}, we simply set K_{ctrl} as a threshold value. This is because the depth of control dependency is often not directly dependent on the method length. We evaluate the sensitivity of our approach to these parameters on all the three tasks with the same experimental setting of CCS2VEC as in **RQ2**.

TABLE V: Sensitivity Analysis Results of $K_{data} = |N|/a + 1$

Setting	Task 1 (BLEU-4)	Taks 2 (F1)	Task 3 (F1)
$a = 1$	22.4	73.3	73.3
$a = 2$	25.5	74.6	74.7
$a = 4$	26.2	78.3	75.6
$a = 8$	20.1	70.9	72.6

TABLE VI: Sensitivity Analysis Results of K_{ctrl}

Setting	Task 1 (BLEU-4)	Taks 2 (F1)	Task 3 (F1)
$K_{ctrl} = 2$	25.2	76.6	72.9
$K_{ctrl} = 3$	26.2	78.3	75.6
$K_{ctrl} = 4$	25.7	74.1	73.0

Sensitivity to K_{data}. We set a to 1, 2, 4 and 8 while fixing K_{ctrl} to 3. The result is shown in Table V. Notice that $a = 1$ indicates there is no limit on data dependency tracking depth. As a increases from 1 to 4, K_{data} decreases, and the performance of CCS2VEC improves in all the three tasks. This is potentially because a long data dependency path might fail to capture local semantics since it may introduce noises from less relevant data dependencies. On the other hand, as a increases from 4 to 8, the performance of CCS2VEC also degrades in all the three tasks. This indicates that a short data dependency path might capture incomplete or fragmented semantics. Therefore, we suggest to set a to 4, which is used in **RQ1** and **RQ2**.

Sensitivity to K_{ctrl}. We set K_{ctrl} to 2, 3 and 4 while fixing a to 4. The result is presented in Table VI. We can observe that as K_{ctrl} increases from 2 to 3, there is a performance improvement in all the three tasks, which potentially owes to a more complete capture of control dependencies. However, as K_{ctrl} increases from 3 to 4, the performance drops in all the three tasks. This may indicate that deeply nested control structures may not have positive impact on prediction but may introduce noises. Hence, we suggest to set K_{ctrl} to 3, which is used in **RQ1** and **RQ2**.

A relatively short or long data and control dependency tracking depth can degrade the performance of CCS2VEC.

G. Threats to Validity

One threat to our evaluation is the size of pre-training data set. However, our data set is already large if compared to prior approaches. We plan to further realize a *large* pre-trained model by incorporating commits from different programming languages. Another threat is the potentially imperfect data set of non-bug-fix commits in **RQ2** and non-defective commits in **RQ3**. However, we actually follow prior approaches to prepare these fine-tuning data sets, and this threat is shared with prior approaches. Data cleaning methods are needed to further improve CCS2VEC.

V. RELATED WORK

Code representation learning techniques have been widely explored to learn code representations from a massive code corpus. They represent code at the level of tokens, AST, CFG, DFG or execution traces, and have been applied to various programming language and software engineering tasks. We refer the readers to surveys [1, 2, 13, 55] for a comprehensive review of learning code representations. Here, we specifically focus our review on code change representation learning and its applications.

Code Change Representation Learning. Tufano et al. [48] used neural machine translation to transform the method before code change to the method after code change. Nie et al. [34] further considered binary file changes. These methods represented the method code at the token level. Different, Qureshi et al. [38] and Lozoya et al. [4] represented the method code at the level of AST paths, and learned a vector representation of code changes. Lozoya et al. [4] only used changed AST paths to represent the source code before and after changes, while Qureshi et al. [38] also included unchanged AST paths. Hoang et al. [18] modeled the hierarchical structure (from words to a line and from lines to a hunk) of code changes with a hierarchical attention network. They represented removed and added code at the token level for supporting unparseable code, and learned the differences between removed and added code as vectors. Different from these approaches, Brody et al. [3] completed code changes given a partially changed code snippet. They represented code changes as AST paths based on four edit operations (i.e., move, update, insert and delete) on ASTs. These previous approaches model code changes at the lexical or syntactical level, making them difficult to learn code change semantics. In contrast, we use data and control dependencies of code changes to ease the semantic representation and learning of code changes.

Commit Message Generation. Jiang et al. [22] and Loyola et al. [31] represented code changes of a commit as a sequence of tokens of removed and added code. To address the problem of out-of-vocabulary words in the above approaches, Liu et al. [28] and Xu et al. [53] incorporated the copying mechanism [42]. Liu et al. [30] proposed an information retrieval-based approach, which was simpler and faster than the above approaches. Wang et al. [51] tried to combine token-level learning-based approach with information retrieval-based approach. Liu et al. [29] represented code changes of a commit as a sequence of AST paths, and integrated their learning-based approach with information retrieval by hybrid ranking. Dong et al. [11] enriched the ASTs before and after code changes with sub-token information, code edit operations and code sequential information to form a graph representation, and leveraged graph neural network to generate commit messages. These approaches are specifically designed for the single task of commit message generation, while our work can support multiple tasks through fine-tuning. Moreover, they only leverage lexical or syntactical information, but do not consider semantic information like we do.

Commit Classification. Tian et al. [47] defined and extracted features from commit messages and code changes, and learned a binary classification model to identify bug-fix commits. Zafar et al. [59] fine-tuned the pre-trained BERT model to solve the bug-fix commit identification task. However, they only used commit messages, but did not consider code changes. Hoang et al. [19] used CNN to learn a representation of commit messages and 3D-CNN to learn a representation of code changes with hierarchical information (i.e., hunks, lines and words) for identifying bug-fix commits. However, they also target one single task, and do not consider semantic information. Another similar task is security patch identification (i.e., to identify commits that fix security bugs), which is addressed by program analysis [54], machine learning [6, 41, 63] and deep learning [33, 52, 61, 64]. However, these learning-based approaches shared the same limitation with those bug-fix commit identification approaches. It is interesting to fine-tune our model to solve security patch identification.

Just-In-Time Defect Prediction. Yang et al. [57] used deep belief network to generate and integrate advanced features from a set of basic features, and learned a machine learning classifier to predict defective commits. To avoid manually defining features, Hoang et al. [17] represented commit messages and code changes as a sequence of words and code tokens to automatically extract features via deep learning. Pornprasit et al. [36] used the frequency of each code token in a commit as features to learn a classification model, which achieved better performance than deep learning-based approaches. They further ranked defective code lines by LIME [40]. Similarly, Zeng et al. [60] adopted one simple feature (i.e., the number of added lines) with the logistic regression classifier, and outperformed deep learning-based approaches. Zhou et al. [62] tried to combine complex deep learning-based approaches and simple machine learning-based approaches. However, previous approaches lack semantic understanding of code changes, which might hinder their prediction effectiveness. In addition, Pornprasit et al. [35] proposed a local rule-based model-agnostic technique to generate explanations of JIT defect models, while Yan et al. [56] and Qiu et al. [37] took a step further to localize defective lines. These approaches are orthogonal to ours.

Program Repair. Tufano et al. [49] adopted neural machine translation to translate the buggy code to the fixed code. Chen et al. [7] further leveraged the copying mechanism [42] to address the unlimited vocabulary problem. These two approaches represented buggy/fixed code as a sequence of tokens. Lutellier et al. [32] distinguished buggy lines and their surrounding context code so as to learn the representation of the transformation from buggy lines with context to fixed lines. Similarly, Li et al. [25] modeled the context code at the level of ASTs. Jiang et al. [21] leveraged a pre-trained programming language model and fine-tuned Lutellier et al.'s model [32], and designed a code-aware beam search strategy to find more correct fixes. Differently, Ding et al. [10] and Chakraborty and Ray [5] used neural machine translation to generate code edits rather than the whole fixed code. Besides, several approaches leveraged code change representation to predict the correctness of patches in program repair [26, 45, 46]. These approaches also do not consider the semantics under code changes. We plan to fine-tune our model to support program repair related tasks.

VI. Conclusions

We have developed a slice-based code change representation approach which takes into account data and control dependencies. We have also developed a pre-trained sparse Transformer model CCS2VEC to learn code change representations on three pre-training tasks. We have also conducted experiments by fine-tuning our pre-trained model on three downstream tasks, which have demonstrated the improvement of CCS2VEC over the state-of-the-art approaches. The source code of our approach and data set are available at https://ccs2vec.github.io/.

Acknowledgment

This work was supported by the National Key R&D Program of China (2021ZD0112903) and the National Natural Science Foundation of China (61972098). Bihuan Chen is the corresponding author of this paper.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, p. 81, 2018.

[2] P. Bielik, V. Raychev, and M. Vechev, "Programming with" big code": Lessons, techniques and applications," in *SNAPL*, 2015, pp. 1–10.

[3] S. Brody, U. Alon, and E. Yahav, "A structural model for contextual code changes," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[4] R. Cabrera Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *SN Computer Science*, vol. 2, no. 3, pp. 1–16, 2021.

[5] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *ASE*, 2021, pp. 443–455.

[6] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A machine learning approach for vulnerability curation," in *MSR*, 2020, pp. 32–42.

[7] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[8] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019, pp. 4171–4186.

[10] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: the data and the metaphor," in *ASE*, 2020, pp. 275–286.

[11] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "Fira: Fine-grained graph-based code change representation for automated commit message generation," in *ICSE*, 2022.

[12] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *ICSE*, 2013, pp. 422–431.

[13] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *SNAPL*, 2017, pp. 1–14.

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

[15] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*, 2021.

[16] K. Herzig and A. Zeller, "The impact of tangled code changes," in *MSR*, 2013, pp. 121–130.

[17] 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 *MSR*, 2019, pp. 34–45.

[18] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *ICSE*, 2020, p. 518–529.

[19] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2471–2486, 2021.

[20] M. Ilse, J. Tomczak, and M. Welling, "Attention-based deep multiple instance learning," in *ICML*, 2018, pp. 2127–2136.

[21] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *ICSE*, 2021, pp. 1161–1173.

[22] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *ASE*, 2017, pp. 135–146.

[23] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *ICSE*, 2011, p. 351–360.

[24] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *ACL*, 2020, pp. 7871–7880.

[25] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *ICSE*, 2020, pp. 602–614.

[26] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 3, pp. 1–29, 2022.

[27] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *arXiv preprint arXiv:2106.04554*, 2021.

[28] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, "Generating commit messages from diffs using pointer-generator network," in *MSR*, 2019, pp. 299–309.

[29] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1800–1817, 2022.

[30] 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 *ASE*, 2018, pp. 373–384.

[31] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *ACL*, 2017, pp. 287–292.

[32] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *ISSTA*, 2020, pp. 101–114.

[33] G. Nguyen-Truong, H. J. Kang, D. Lo, A. Sharma, A. E. Santosa, A. Sharma, and M. Y. Ang, "Hermes: Using commit-issue linking to detect vulnerability-fixing commits," in *SANER*, 2022, pp. 51–62.

[34] L. Y. Nie, C. Gao, Z. Zhong, W. Lam, Y. Liu, and Z. Xu, "Coregen: Contextualized code representation learning for commit message generation," *Neurocomputing*, vol. 459, pp. 97–107, 2021.

[35] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, "Pyexplainer: Explaining the predictions of just-in-time defect models," in *ASE*, 2021, pp. 407–418.

[36] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *MSR*, 2021, pp. 369–379.

[37] F. Qiu, Z. Gao, X. Xia, D. Lo, J. Grundy, and X. Wang, "Deep just-in-time defect localization," *IEEE Transactions on Software Engineering*, 2021.

[38] S. A. Qureshi, S. Mehta, R. Bhagwan, and R. Kumar, "Assessing the effectiveness of syntactic structure to learn code edit representations," *arXiv preprint arXiv:2106.06110*, 2021.

[39] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *ICSE*, 2016, p. 428–439.

[40] M. T. Ribeiro, S. Singh, and C. Guestrin, """ why should i trust you?" explaining the predictions of any classifier," in *KDD*, 2016, pp. 1135–1144.

[41] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *ICSME*, 2018, pp. 579–582.

[42] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *ACL*, 2017, pp. 1073–1083.

[43] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *NAACL*, 2018, pp. 464–468.

[44] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *FSE*, 2012, pp. 1–11.

[45] H. Tian, Y. Li, W. Pian, A. K. Kabore, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, pp. 1–30, 2022.

[46] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *ASE*, 2020, pp. 981–992.

[47] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE*, 2012, pp. 386–396.

[48] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation,"

in *ICSE*, 2019, pp. 25–36.

[49] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *ASE*, 2018, pp. 832–837.

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017, pp. 5998–6008.

[51] H. Wang, X. Xia, D. Lo, Q. He, X. Wang, and J. Grundy, "Context-aware retrieval-based deep commit message generation," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 4, pp. 1–30, 2021.

[52] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[53] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *IJCAI*, 2019, pp. 3975–3981.

[54] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *ICSE*, 2017, pp. 462–472.

[55] E. Yahav, "Programming with "big code"," in *APLAS*, 2015, pp. 3–8.

[56] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 82–101, 2022.

[57] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *QRS*, 2015, pp. 17–26.

[58] Z. Ye, Q. Guo, Q. Gan, X. Qiu, and Z. Zhang, "Bp-transformer: Modelling long-range context via binary partitioning," *arXiv preprint arXiv:1911.04070*, 2019.

[59] S. Zafar, M. Z. Malik, and G. S. Walia, "Towards standardizing and improving classification of bug-fix commits," in *ESEM*, 2019, pp. 1–6.

[60] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *ISSTA*, 2021, pp. 427–438.

[61] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *ASE*, 2021, pp. 705–716.

[62] X. Zhou, D. Han, and D. Lo, "Simple or complex? together for a more accurate just-in-time defect predictor," in *ICPC*, 2022, pp. 229–240.

[63] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *ESEC/FSE*, 2017, pp. 914–919.

[64] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–27, 2021.