Xu SB, Chen SY, Yao Y *et al*. Detecting and untangling composite commits via attributed graph modeling. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, 40(1): 119-137, Jan. 2025. DOI: 10.1007/s11390-024-2943-9, CSTR: 32374.14.s11390-024-2943-9

Detecting and Untangling Composite Commits via Attributed Graph Modeling

Sheng-Bin Xu (徐圣斌), Si-Yu Chen (陈思宇), Yuan Yao* (姚　远), *Member, CCF*
and Feng Xu (徐　锋), *Member, CCF*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

E-mail: kingxu@smail.nju.edu.cn; mf20330009@smail.nju.edu.cn; y.yao@nju.edu.cn; xf@nju.edu.cn

Received November 21, 2022; accepted April 30, 2024.

Abstract During software development, developers tend to tangle multiple concerns into a single commit, resulting in many composite commits. This paper studies the problem of detecting and untangling composite commits, so as to improve the maintainability and understandability of software. Our approach is built upon the observation that both the textual content of code statements and the dependencies between code statements are helpful in comprehending the code commit. Based on this observation, we first construct an attributed graph for each commit, where code statements and various code dependencies are modeled as nodes and edges, respectively, and the textual bodies of code statements are maintained as node attributes. Based on the attributed graph, we propose graph-based learning algorithms that first detect whether the given commit is a composite commit, and then untangle the composite commit into atomic ones. We evaluate our approach on nine C# projects, and the results demonstrate the effectiveness and efficiency of our approach.

Keywords composite commit, commit untangling, code dependency graph, attributed graph

1 Introduction

During software development, developers often use version control systems (e.g., Git or SVN) to track the history of software changes in the form of commits. A commit typically consists of two parts: 1) code diffs, which are the changes to the source code between the previous version and the current version; 2) a message, which is usually a textual description that summarizes the content and intent of the code changes. A commit is called atomic if it represents one distinct concern (such as fixing a bug or adding a new feature), and there are many benefits (e.g., easy code review and reuse) to making each commit atomic as suggested by existing research[1–3].

However, existing studies have empirically found that composite commits (e.g., adding a new feature and fixing a bug in the same commit) are very common in both enterprise and open source projects[4–7], making it necessary to automatically detect and untangle such commits. Researchers have proposed a variety of approaches to decompose composite commits into atomic ones[6, 8–14]. Among them, the state-of-the-art approaches, i.e., FLEXEME[13] and SMARTCOMMIT[14], propose to build a dependency graph for each commit and then adopt graph clustering algorithms to untangle composite commits.

However, recent approaches still suffer from several limitations. SMARTCOMMIT constructs the graph by using diff hunks (i.e., the adjacent code change statements) as nodes and dependencies as edges, and clusters nodes based on edge connectivity. However, diff hunks corresponding to different concerns may have strong edge connectivity, making them difficult to be split by SMARTCOMMIT. FLEXEME uses each code statement as a graph node, and clusters nodes by cal-

Regular Paper

A preliminary version of the paper was published in the Proceedings of Internetware 2022.

This work was supported by the National Natural Science Foundation of China under Grant No. 62025202 and the Fundamental Research Funds for the Central Universities under Grant No. 020214380102.

*Corresponding Author

culating the similarity between the first-order-neighbor subgraphs for two changed statements. However, the first-order neighbors cannot capture the global information well, especially when the graph is relatively large. Additionally, existing approaches mainly focus on untangling the given commit without checking if it is a composite commit, bearing the risk of further splitting an atomic commit into meaningless pieces. Moreover, existing proposals primarily consider clustering the code into atomic commits only using the graph structure and disregard the naturalness of software[15]. That is, code is readable to humans to some extent because it contains semantic information (in a manner similar to natural languages) that can aid in better comprehending the commit.

To address the limitations of existing work, in this paper, we propose a new approach that systematically considers both detecting and untangling composite commits. Specifically, our approach consists of three stages.

- *Stage* 1*: Commit Graph Construction*. We first construct a commit graph for each commit. In this commit graph, we consider each code statement as a node, and incorporate various edges related to the control flow, data flow, and name flow in the code. We further keep the code statement itself as the attribute of each node. Based on our graph construction, code in the same diff hunk can be separated into multiple atomic commits as the graph is built on the statement level. Moreover, the naturalness of the code is preserved in node attributes.
- *Stage* 2*: Composite Commit Detection*. Next, before a commit is fed into an untangling module, a neural network classifier ComDet is trained upon the constructed commit graph to detect the composite commits. The proposed ComDet uses graph neural networks to encode both the node attributes and the graph structure into node embeddings, and then adopts a novel global context-aware attention to aggregate node embeddings into a graph embedding. This graph embedding is used to predict whether the given commit is a composite one or not.
- *Stage* 3*: Composite Commit Untangling*. Finally, a graph clustering algorithm called ComUnt is designed to untangle code changes into atomic commits. Similar to ComDet, ComUnt handles both the graph structure and node attributes. Different from the existing work, ComUnt adopts the high-order graph convolution[16] so as to capture long-range dependencies, and automatically adjusts the number of clusters through affinity propagation[17], as the number of atomic commits in a composite commit is unknown in advance.

To evaluate the proposed approach, we use the dataset provided by Flexeme[13], which contains identified atomic commits and carefully synthesized composite commits from nine C# projects on GitHub. The evaluation results show that: 1) ComDet achieves a significant relative improvement of up to 89.7% in detection accuracy (F1-score) compared with the baseline; 2) the untangling accuracy of ComUnt is 7.8% higher than that of Flexeme, and ComUnt runs more than six times faster; and 3) by combining ComDet and ComUnt, we can achieve up to 8.2% higher accuracy than Flexeme.

In summary, this paper makes the following contributions:

- a commit untangling framework that first detects composite commits and then untangles them,
- an attributed commit graph construction method that considers both the code dependencies and the code naturalness,
- a new approach for detecting composite commits based on graph neural networks,
- the deep graph clustering algorithm ComUnt that is specially designed for commit untangling,
- experimental evaluations showing that the proposed approach outperforms its competitors.

This work is an extension of our prior conference version[18]. In this extension, we 1) redesign the commit untangling workflow by adding a composite commit detection module to proactively identify atomic commits and prevent their further decomposition, and 2) design three more research questions with experimental results supporting the effectiveness of the composite commit detection module and the entire commit untangling framework as a whole. The rest of the paper is organized as follows. Section 2 presents the necessary background knowledge, and Section 3 provides a motivational example. Section 4 describes the proposed approach. Section 5 presents the evaluation setup, and Section 6 shows the evaluation results. Section 7 discusses the threats to validity, and Section 8 covers the related work. Section 9 concludes the paper.

2 Background

In this section, we introduce the background knowledge, including composite commits, graph modeling of code, graph neural networks, and graph clustering.

2.1 Composite Commits

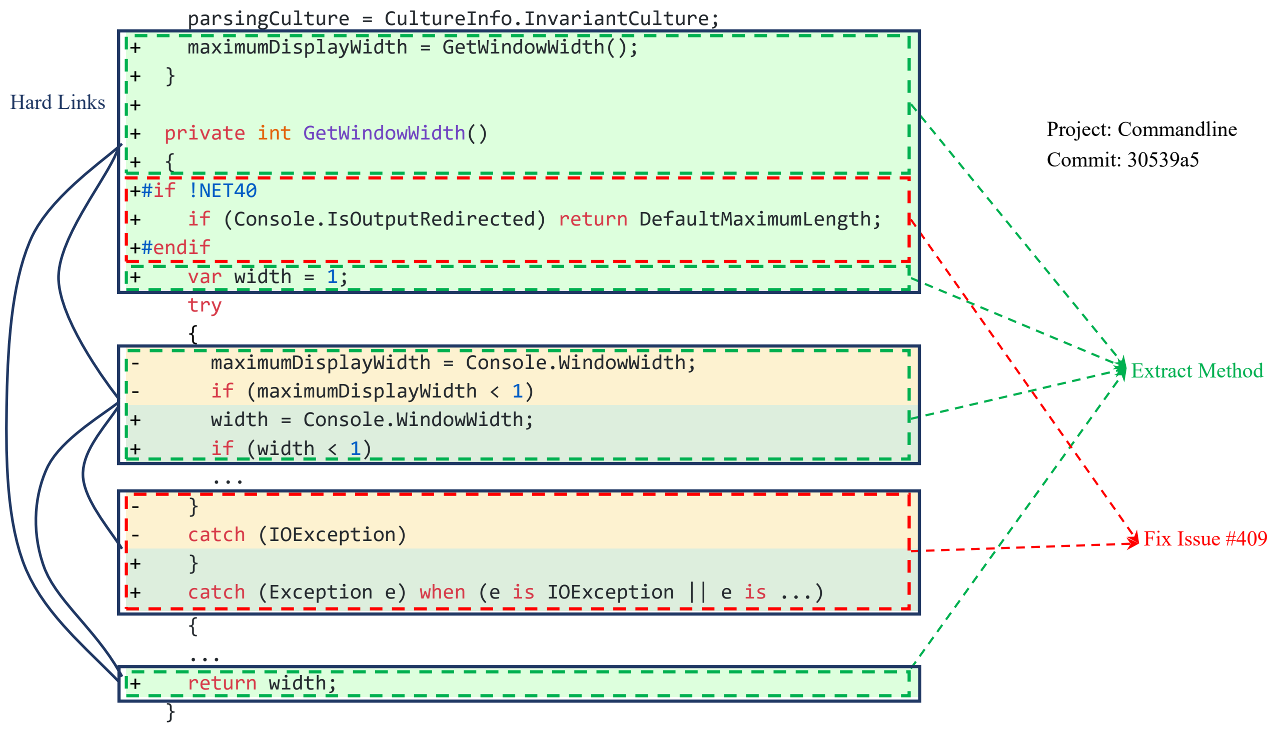
In software development, a commit is considered as a composite commit if it involves multiple development activities, e.g., fixing a bug or adding a new feature. In Fig.1, we show an example of a real composite commit from the Commandline project[①]. This commit contains two activities which can be untangled, including code refactoring (i.e., the changes in the green dotted boxes are extracting method refactoring) and bug fixing (i.e., the changes in the red dotted boxes are fixing issue #409). The goal of this work is to first detect such composite commits and further decompose each of them into atomic ones, so as to make the code easier for developers to review and reuse.

2.2 Graph Modeling of Code

Recently, researchers have tended to model a piece of code as a dependency graph to learn its semantics for various software engineering tasks[19–24]. Commonly considered dependencies include abstract syntax trees (ASTs), control flow graphs (CFGs)[25], data flow graphs (DFGs)[26], and so on. Aside from the three types of dependencies mentioned above, there also exist call graphs (representing the invoking relationship between methods in the program), class inheritance graphs (representing the extending relationship between classes in the program), and others. Furthermore, researchers also propose merging different types of graphs. Common examples include program dependency graphs (merged by CFG and DFG)[27] and code property graphs (merged by AST, CFG, and DFG)[28].

2.3 Graph Neural Networks

Graph neural networks (GNNs)[29–31] have been successfully applied to a variety of graph data analysis tasks. In general, GNNs use the graph structure and node features to learn a representation vector for each node. Specifically, GNNs follow a neighborhood aggregation strategy, where each node iteratively updates its representation by aggregating the representations of its neighbors. After k iterations, the representation of each node captures the structural information within its k-hop neighbors. Different types of



```
      parsingCulture = CultureInfo.InvariantCulture;
+     maximumDisplayWidth = GetWindowWidth();
+   }
+
+   private int GetWindowWidth()
+   {
+#if !NET40
+     if (Console.IsOutputRedirected) return DefaultMaximumLength;
+#endif
+     var width = 1;
      try
      {
-       maximumDisplayWidth = Console.WindowWidth;
-       if (maximumDisplayWidth < 1)
+       width = Console.WindowWidth;
+       if (width < 1)
        ...
-     }
-     catch (IOException)
+     }
+     catch (Exception e) when (e is IOException || e is ...)
      {
      ...
+     return width;
    }
```

Fig.1. Example of a real composite commit. SmartCommit cannot successfully untangle this composite commit due to the hard links (the blue curves on the left) between diff hunks (the changes within solid-line boxes). In contrast, our approach treats each code statement as a node and uses the deep graph clustering algorithm, which makes it more flexible and powerful.

①https://github.com/commandlineparser/commandline/commit/30539a51285735bf906a0ae350750982ddb28691, Nov. 2024.

GNNs use different aggregation operations. For example, graph convolutional networks (GCNs)[29] adopt the convolution operation and gated graph neural networks (GGNNs)[30] use gating mechanism.

2.4 Graph Clustering

Graph clustering aims to divide the nodes of a graph into different groups[17], so that the nodes within the same group are more similar/related and the nodes from different groups are more different/irrelated. Traditional graph clustering algorithms can be divided into flow-based algorithms and spectrum-based algorithms[17]. The flow-based algorithms generally solve the maximum-flow and minimum-cut problems[32–34]. The spectrum-based algorithms typically compute node similarities by combining several eigenvectors derived from the Laplacian matrix of graphs[35, 36]. Recent progress has applied deep learning to graph clustering, which is called deep graph clustering[37, 38]. One of the main reasons for this progress is the rapid development of graph neural networks.

3 Motivation

In this section, we explain the limitations of two state-of-the-art commit untangling approaches, SmartCommit and Flexeme.

3.1 Limitations of SmartCommit

As mentioned before, SmartCommit[14] builds a graph from the composite commit, using diff hunks (which are defined based on the proximity of multiple changed statements) as the nodes. SmartCommit further defines several types of edges, and one of them is named "hard link". Specifically, if any version of the code in a diff hunk has a call relationship, data dependency, or AST node connection with another diff hunk, there is a hard link between these two diff hunks. SmartCommit assigns the largest weight to such hard links, making diff hunks with hard links inseparable. An example is shown in Fig.1, where the composite commit consists of two atomic commits (marked with green dotted boxes and red dotted boxes, respectively). However, SmartCommit cannot make the right division as all diff hunks are connected with the hard links. This strongly limits the applicability of SmartCommit. For example, in practical bug fixing activities, developers often refactor code in close proximity, which generates hard links between the bug fixing hunks and the refactoring hunks.

Another potential weakness of SmartCommit is that it cannot split the changed statements inside a diff hunk. That is, if some of the changed statements within the same diff hunk belong to different atomic commits, SmartCommit cannot split them.

3.2 Limitations of Flexeme

Flexeme[13] builds a δ-NFG from the composite commit, calculates the similarity of the k-hop-neighbor subgraph between two nodes using the Weisfeiler-Lehman graph kernel[39], and applies the standard agglomerative clustering to obtain atomic commits. The δ-NFG is a δ-PDG augmented with name flows, and the δ-PDG is a combination of multiple versions of program dependency graphs. Each graph node in Flexeme corresponds to a code statement. This statement-level granularity enables Flexeme to effectively decompose the composite commit at the statement level, thereby overcoming SmartCommit's limitation of being constrained by the original diff hunk boundaries.

However, there are still several limitations of Flexeme. First, the high time complexity of graph-kernel based similarity computation makes it difficult to scale to large graphs. Thus, Flexeme only uses the first-order-neighbor subgraph for similarity calculation, and this makes the similarity results constrained in a local area without capturing the global knowledge of the composite commit. This affects the performance, especially when the composite commit is relatively large (e.g., containing thousands of nodes)[②]. Second, although the adopted agglomerative clustering of Flexeme does not need to pre-specify the number of clusters, it still needs to set a sensitive threshold parameter to decide when the clustering terminates[③]. Third, the code itself contains rich semantic information from the readability/naturalness aspect, and Flexeme only considers name flows (two statements containing the same identifier) with-

[②]This claim is also supported by our experimental results in Subsection 6.4.

[③]For example, we find that the clustering results of 47.5% composite commits changed after changing the threshold from the default 0.5 to 0.6.

out taking full advantage of, e.g., the subtokens in the identifier.

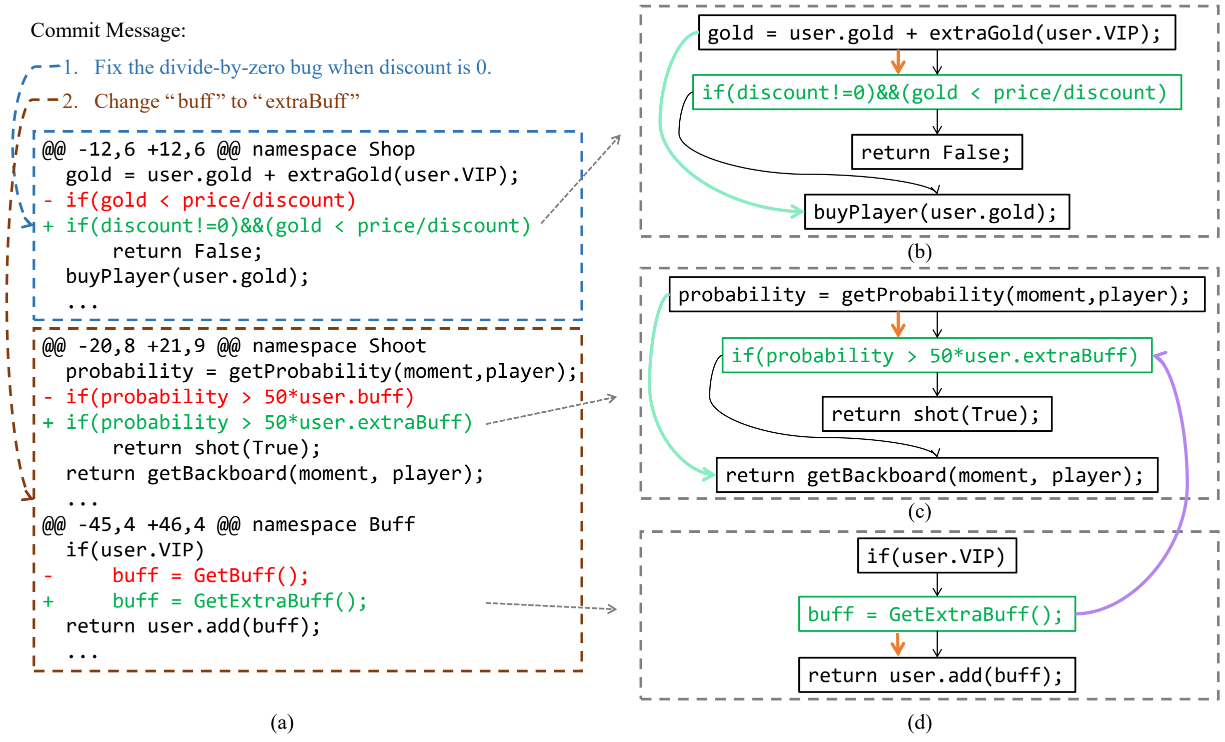
An illustrative example of the third limitation is shown in Fig.2. This commit has two goals: fixing a divided-by-zero bug, and renaming "buff" to "extraBuff". Intuitively, this composite commit should be untangled into two atomic commits accordingly, as indicated by the dotted boxes. Figs.2(b)–2(d) show the three corresponding subgraphs of the full δ-NFG constructed by FLEXEME. For simplicity, we only draw the node representing the statement after the change. As FLEXEME's graph similarity computation is based on the first-order neighbors, the three changed statements (in green boxes) cannot be merged due to the label mismatch of their first-order neighbors. In contrast, COMUNT utilizes the textual embeddings of code statements and adds a new edge type, i.e., subtoken co-occurrence (the purple edge in Fig.2). In this way, COMUNT makes the changed statements in Fig.2(c) and Fig.2(d) closer to each other instead of being easily separated.

4 Proposed Approach

In this section, we present the proposed approach. Fig.3 shows an overview of our approach, which consists of three stages: commit graph construction, composite commit detection, and composite commit untangling. Specifically, in the commit graph construction stage, we build an attributed commit graph for each commit. Nodes in the commit graph correspond to both changed code statements and unchanged code statements related to the commit. The edges are inserted based on various code dependencies, such as control flows and data flows. In the second stage of composite commit detection, we train a classifier upon the constructed commit graph to detect composite commits. The detected composite commits are fed into the next stage. Finally, in the composite commit untangling process, we cluster each detected composite commit into atomic commits. The clustering is applied to the changed nodes (nodes corresponding to changed code statements) of the commit graph, and each resulting cluster is treated as an atomic commit. In Subsections 4.1–4.3, we will describe these three stages in detail.

4.1 Commit Graph Construction

A commit graph is defined as $G = (V, E)$. V is the set of nodes and each node corresponds to a code statement in the form of $< code, function, sign >$. Here, *code* represents the body of the code statement, and it is also used as the node attribute; *function* is


Commit Message:
1. Fix the divide-by-zero bug when discount is 0.
2. Change "buff" to "extraBuff"

```
@@ -12,6 +12,6 @@ namespace Shop
  gold = user.gold + extraGold(user.VIP);
- if(gold < price/discount)
+ if(discount!=0)&&(gold < price/discount)
      return False;
  buyPlayer(user.gold);
  ...
@@ -20,8 +21,9 @@ namespace Shoot
  probability = getProbability(moment,player);
- if(probability > 50*user.buff)
+ if(probability > 50*user.extraBuff)
      return shot(True);
  return getBackboard(moment, player);
  ...
@@ -45,4 +46,4 @@ namespace Buff
  if(user.VIP)
-     buff = GetBuff();
+     buff = GetExtraBuff();
  return user.add(buff);
  ...
```

(a)

```
gold = user.gold + extraGold(user.VIP);
if(discount!=0)&&(gold < price/discount)
return False;
buyPlayer(user.gold);
```

(b)

```
probability = getProbability(moment,player);
if(probability > 50*user.extraBuff)
return shot(True);
return getBackboard(moment, player);
```

(c)

```
if(user.VIP)
buff = GetExtraBuff();
return user.add(buff);
```

(d)

Fig.2. Motivation example[18]. (a) Composite commit that should be untangled into two atomic commits. (b)-(d) Subgraphs of the δ-NFG constructed by FLEXEME.

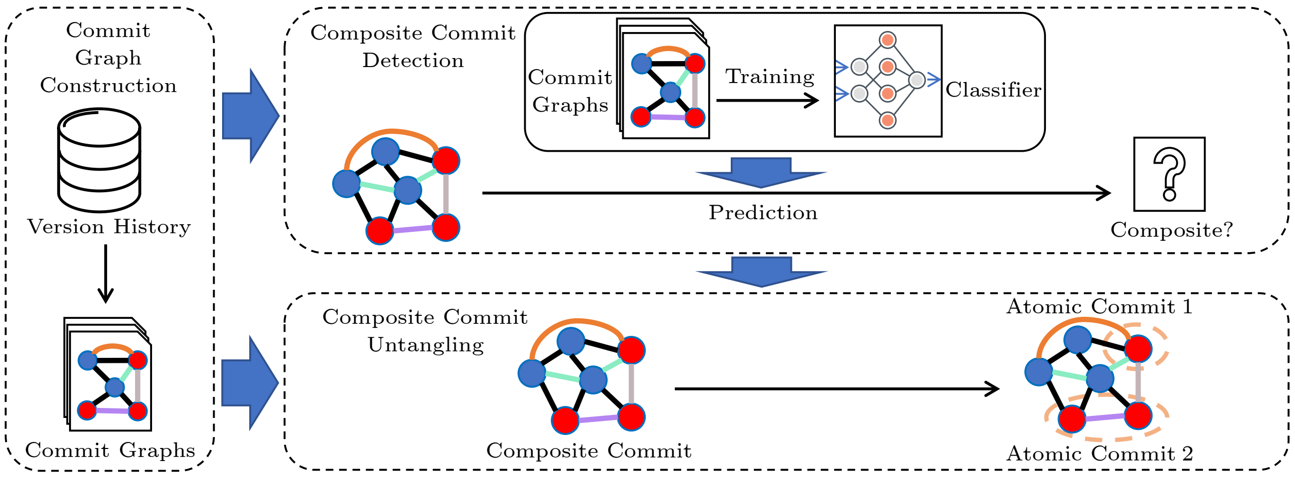


Fig.3. Overview of our approach.

the signature of the method to which the code statement belongs, and it helps to merge the two versions of code before and after the commit; *sign* is a Boolean value indicating whether the code statement is a changed statement or not.

E is the set of edges, and each edge is in the form of (v_i, r, v_j) where r stands for the edge type between nodes v_i and v_j. Specifically, for both the versions before and after the commit, we first build the CFGs for them. On the basis of the CFGs, we use four kinds of information to augment them: call graph edges, data flows, name flows[40], and subtoken co-occurrence. The call graph edges represent the invocation relationships between methods in the program. The data flows show the flows of data through the program's entire execution. The name flows are built upon the data flows, and they collect the names whose values flow into a given variable. We follow [40] to insert the name flow edges. A subtoken edge is inserted between two nodes if they contain variables or methods with any shared subtokens. For example, for the two methods "openBox( )" and "closeBox( )", we split them into subtokens, delete some common prefixes (e.g., set, get, on), and obtain "open box" and "close box" from them. Then, a subtoken edge is inserted between the corresponding code statements since the subtoken "box" co-occurs.

To obtain the final commit graph for a commit, we further merge the two graphs of the versions before and after the commit. Since exactly two identical code statements are rarely found in the same method, we merge two nodes into one node if their *code* and *function* are identical. We reserve the remaining nodes. For edges, we first keep all edges from both versions, and then merge the edges that share the same source and target nodes with the same edge type into one.

An example of this commit graph construction process is shown in Fig.4. For simplicity, we only show the *code* attribute of the node (we do not show *function* and *sign*). This code snippet in the example is to give a player a gift box. In this refactoring commit, "player" is renamed to "user". The related statements will also need to be changed. We first build CFGs (represented by black edges) for both versions (denoted as Version 1 and Version 2 in Fig.4), and then augment them with call graph edges (represented by gray edges), data flow edges (represented by orange edges), name flow edges (represented by green edges), and subtoken edges (represented by purple edges). Note that although name flow edges and subtoken edges are the same in the figure, they are inserted for different reasons. The name flow edges arise from the def-use relationship of the variable "box", and subtoken edges result from the co-occurrence of the subtoken "box" in "createBox" and "openBox". Then, we merge the two graphs to form the final commit graph. We first merge the nodes with the same code statements, i.e., the middle three code statements, and also merge the edges accordingly. Then, we keep the rest of the nodes and edges, as shown in Fig.4(c).

4.2 Composite Commit Detection

As shown in Fig.3, the detector ComDet is a neural network classifier and is responsible for predicting whether a commit is composite or not. It is trained in a supervised manner, taking a commit graph as the input and estimating the probability that this com-

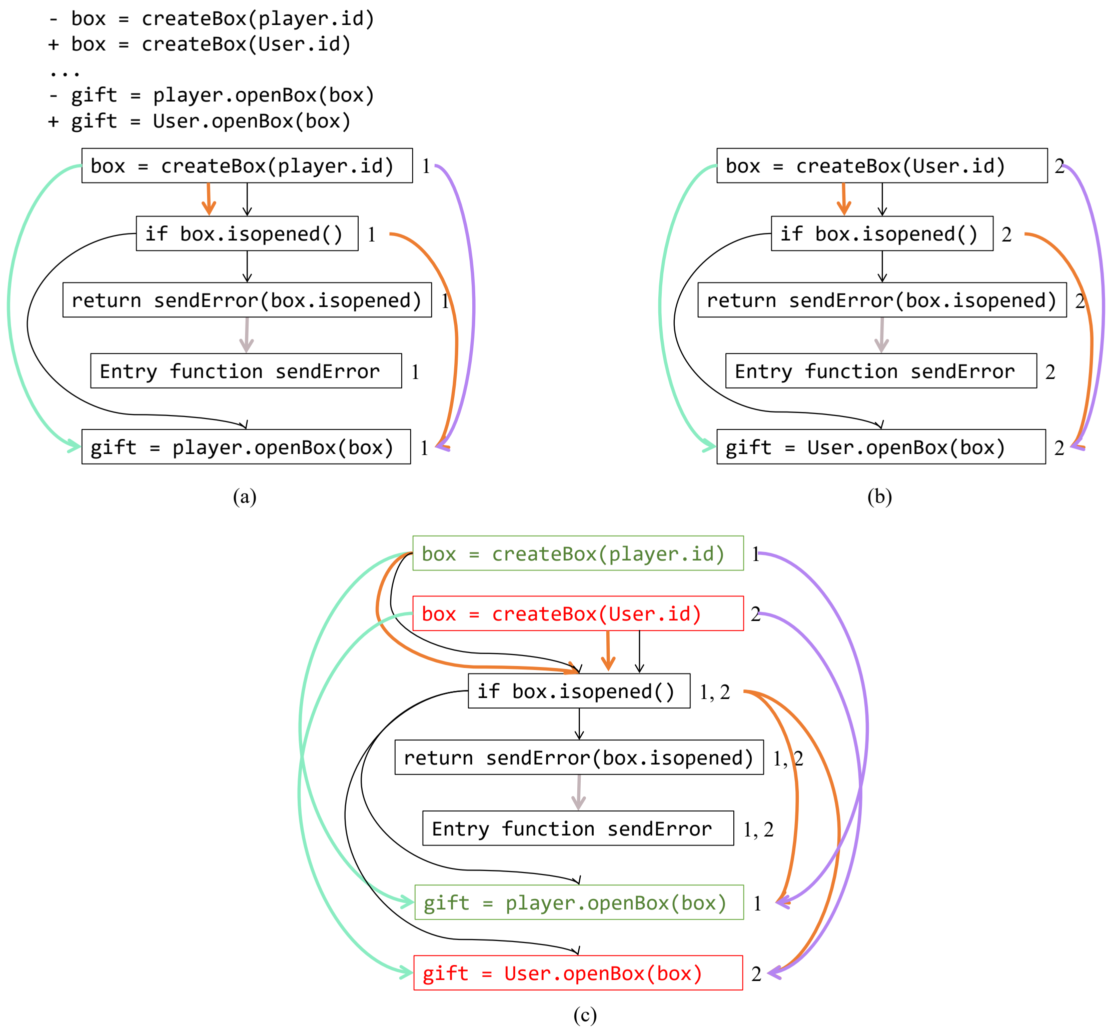

- box = createBox(player.id)
+ box = createBox(User.id)
...
- gift = player.openBox(box)
+ gift = User.openBox(box)
box = createBox(player.id)
if box.isopened()
return sendError(box.isopened)
Entry function sendError
gift = player.openBox(box)
(a)
box = createBox(User.id)
gift = User.openBox(box)
(b)
1, 2
(c)

Fig.4. Construction of a commit graph[18]. (a) Graph of Version 1. (b) Graph of Version 2. (c) Commit graph. Each node stands for a code statement. Edges stand for various code dependencies with black, gray, orange, green, and purple representing the control flows, call graph edges, data flows, name flows, and subtoken co-occurrence, respectively.

mit is composite. Specifically, ComDet uses the gated graph neural network (GGNN)[30] to learn the latent embedding of each node in the commit graph, and then uses an attention layer to aggregate the embeddings of all nodes to form the graph representation. The graph representation is fed into a fully-connected layer for predicting the probability.

GGNN Modeling of Commit Graph. For a commit graph $G=(V, E)$, we denote the initial vector representation of node $v_i \in V$ as $\boldsymbol{s}_i$. $\boldsymbol{s}_i$ is first fed into a fully-connected layer with the tanh activation function, which aims to fuse the initial features in $\boldsymbol{s}_i$ and set the dimension of the vector to a suitable size,

$$\boldsymbol{s}'_i = \tanh\left(\boldsymbol{W}^{(s)}\boldsymbol{s}_i + \boldsymbol{b}^{(s)}\right), \qquad (1)$$

where $\boldsymbol{W}^{(s)}$ and $\boldsymbol{b}^{(s)}$ are learnable parameters.

Then, a GGNN is used to encode the commit graph G. The GGNN models relationships (edges) between nodes with message passing operations. Specifically, in the l-th layer of the GGNN, the message sent by each node $v_i \in V$ to its neighbors through the edges of type r is computed through a fully-connected layer,

$$\boldsymbol{m}^{(l)}_{r,\ i} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)}_i + \boldsymbol{b}^{(l)},$$

where $\boldsymbol{h}^{(l-1)}_i$ is the hidden state of node v_i from layer $l-1$, and the hidden state of node v_i input to the first layer is $\boldsymbol{s}'_i$. Next, node v_i aggregates all the incoming messages from its neighbors and updates its own state with a gated recurrent unit (GRU),

$$\boldsymbol{m}^{(l)}_i = \sum_{(v_j,\ r,\ v_i)\in E} \boldsymbol{m}^{(l)}_{r,\ j},$$
$$\boldsymbol{h}^{(l)}_i = GRU\left(\boldsymbol{m}^{(l)}_i,\ \boldsymbol{h}^{(l-1)}_i\right).$$

It is worth mentioning that we add the corresponding reverse edges for the directed edges (i.e., control flow edges, call graph edges, data flow edges, and name flow edges) to enable bidirectional message passing in commit graphs. In other words, the GGNN handles a total of nine types of edges. We further concatenate $\boldsymbol{s}_i'$ and $\boldsymbol{h}_i^{(t)}$ (the hidden state of the last GGNN layer) to obtain the final state/embedding of node v_i,

$$\boldsymbol{h}_i = \left[\boldsymbol{s}_i';\ \boldsymbol{h}_i^{(t)}\right],$$

where both the initial features and the features after message passing are preserved.

Global Context-Aware Attention. Next, the node embeddings in a graph are aggregated through global context-aware attention. To generate one embedding per graph using a bag of node embeddings $\boldsymbol{H} = \{\boldsymbol{h}_1, \dots, \boldsymbol{h}_n\}$, one can either perform an unweighted average of the node embeddings, or a weighted sum where the weight associated with one node is determined by its embedding. However, which nodes are more important and receive more weight should depend on the specified similarity metric. Thus, we use the following attention mechanism to let the model learn weights guided by the similarity between nodes and the global graph context.

The global graph context $\boldsymbol{c}$ is derived by applying a nonlinear transformation to the average of node embeddings,

$$\boldsymbol{c} = \tanh\left(\boldsymbol{W}^{(c)}\left(\frac{1}{n}\sum_{1}^{n}\boldsymbol{h}_i\right) + \boldsymbol{b}^{(c)}\right),$$

and it provides the global information of the graph. Based on $\boldsymbol{c}$, we can compute the attention weight for each node in the graph as follows:

$$\alpha_i = \frac{\exp\left(\boldsymbol{h}_i^{\mathrm{T}} \cdot \boldsymbol{c}\right)}{\sum_{j=1}^{n}\exp\left(\boldsymbol{h}_j^{\mathrm{T}} \cdot \boldsymbol{c}\right)}.$$

The aggregated vector $\boldsymbol{e}$ is a linear combination of $\boldsymbol{H}$:

$$\boldsymbol{e} = \sum_{i=1}^{n}\alpha_i \cdot \boldsymbol{h}_i.$$

A noteworthy point is that we only aggregate the vectors of the nodes corresponding to the changed statements as the features corresponding to the unchanged statements have been passed to the changed statements through the message passing mechanism.

The commit graph embedding $\boldsymbol{e}$ is then fed into a fully-connected layer with sigmoid activation:

$$p = \mathrm{sigmoid}\left(\boldsymbol{W}\boldsymbol{e} + \mathrm{b}\right),$$

where p denotes the probability that the commit is composite.

Initialization. The remaining problem is to set the initial node vector $\boldsymbol{s}_i$. We try three different types of representation, i.e.,the TF-IDF vector, the BERT[41] vector, and the CodeBERT[42] vector. For the TF-IDF vector, we treat *code* and *function* in each node as a document and combine the TF-IDF scores of the words in the document fo form a vector. To reduce the vocabulary size and improve the generalizability of the representation, we split the identifier in *code* and *function* into sub-tokens according to the name convention and lowercase all the tokens (e.g.,"getBox" to "get box", "CODE_INDENT" to "code indent"). For the latter two, we use the pre-trained models provided in their original papers to generate both the BERT vector and the CodeBERT vector for each node. To further elaborate, both *code* and *function* are encoded as vectors using the pre-trained models, and then the two vectors are concatenated together to form a node vector.

4.3 Composite Commit Untangling

When a commit is predicted to be composite, we take a bottom-up approach ComUnt to untangle it based on its commit graph. An intuitive idea is that if two changed statements are closely related in the commit graph, they are likely to belong to the same atomic commit of a specific activity. Therefore, by assuming that each code change is atomic in the beginning, we first update the features of each node from the commit graph, and then cluster the nodes of changed statements based on the similarity computed from the updated node features.

Capturing Long-Range Dependency via Graph Convolution. We adopt the graph convolution operator in graph neural networks to update the node features. One advantage of this operator is that it can handle both node attributes and the graph structure by propagating node features through the graph structure. Specifically, the graph convolution can be defined as[16, 29]:

$$\overline{\boldsymbol{x}} = \boldsymbol{F}\boldsymbol{x},$$

where $\boldsymbol{x}$ is the graph signal and $\boldsymbol{F}$ is the graph filter. The graph filter F can be further defined as $\boldsymbol{F} = \boldsymbol{U}g(\boldsymbol{\Lambda})\boldsymbol{U}^{-1}$. Here $\boldsymbol{U}$ and $\boldsymbol{\Lambda}$ are obtained through the eigen-decomposition of the normalized graph

Laplacian $\boldsymbol{L}$, i.e., $\boldsymbol{L}=\boldsymbol{U\Lambda U}^{-1}$, where $\boldsymbol{L}$ is computed as $\boldsymbol{L}=\boldsymbol{I}-\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$, with $\boldsymbol{A}$ denoting the adjacency matrix of the graph, and $\boldsymbol{D}$ denoting the degree matrix of $\boldsymbol{A}$. $\boldsymbol{\Lambda}=\mathrm{diag}(\lambda_1,\ \dots,\ \lambda_n)$ is a diagonal matrix containing the eigenvalues. Function $g(\boldsymbol{\Lambda})=\mathrm{diag}(\mathrm{g}(\lambda_1),\ \dots,\ \mathrm{g}(\lambda_\mathrm{n}))$ is called the frequency response function. A graph signal can be represented as a linear combination of the eigenvectors, i.e.,

$$\boldsymbol{x}=\boldsymbol{Uz}=\sum_{q=1}^{n} z_q\boldsymbol{u}_q,$$

where $\boldsymbol{U}=(\boldsymbol{u}_1,\ \dots,\ \boldsymbol{u}_n)$, $\boldsymbol{z}=(z_1,\ \dots,\ z_n)^{\mathrm{T}}$, and z_q is the coefficient of $\boldsymbol{u}_q$. To preserve $\boldsymbol{F}$ to be a low-pass filter, which can ensure the smoothness of the graph signal for better clustering, we define

$$g(\lambda_q)=1-\frac{1}{2}\lambda_q.$$

Putting the above together, the graph filter $\boldsymbol{F}$ can be written as

$$\boldsymbol{F}=\boldsymbol{U}g(\boldsymbol{\Lambda})\boldsymbol{U}^{-1}=\boldsymbol{U}\left(\boldsymbol{I}-\frac{1}{2}\boldsymbol{\Lambda}\right)\boldsymbol{U}^{-1}=\boldsymbol{I}-\frac{1}{2}\boldsymbol{L},$$

and the formula of updating feature matrix $\boldsymbol{X}$ is

$$\overline{\boldsymbol{X}}=\left(\boldsymbol{I}-\frac{1}{2}\boldsymbol{L}\right)\boldsymbol{X},$$

where the initial node feature matrix $\boldsymbol{X}$ is obtained by inputting node attributes into the pre-trained BERT[41]. However, the above graph convolution is first-order as it updates each node by aggregating its one-hop neighbors only. To capture long-range neighbors, we extend the graph convolution to a k-order graph convolution as[16]:

$$\overline{\boldsymbol{X}}=\left(\boldsymbol{I}-\frac{1}{2}\boldsymbol{L}\right)^{k}\boldsymbol{X}.$$

With the above graph convolution, we can aggregate information from k-hop neighbors. This is important when the composite commit is complex and the commit graph is relatively large.

Clustering Graph Nodes via Affinity Propagation. Next, we apply the affinity propagation clustering[17] to the updated node features to partition the graph nodes into clusters. Affinity propagation is built on the information transmission between different data points, and it gradually converges to the best clustering results without the need to pre-specify the cluster number. It has three key concepts: similarity, responsibility, and availability. Similarity $s(i,\ m)$ represents the feature similarity between point i and a potential cluster center point m. We treat each graph node as a point and set the similarity as the Euclidean distance between the updated node features. Responsibility $r(i,\ m)$ represents the appropriateness of data point m to be the cluster center of data point i, and availability $a(i,\ m)$ represents the appropriateness of point i to select point m as its cluster center. Responsibility and availability are iteratively updated against each other:

$$r(i,m)=s(i,m)-\max_{m',\ m'\neq m}\{a(i,m')+s(i,m')\},$$
$$a(i,m)=\min\{0,r(m,m)+\sum_{i',\ i'\notin\{i,\ m\}}\max\{0,r(i',m)\}\}.$$

We illustrate responsibility and availability in Fig.5. Intuitively, we can regard the affinity propagation clustering as an election process. Each point is both a voter and a candidate, and some of them will be selected as representatives, i.e., the cluster centers. $s(i,\ m)$ can be regarded as the initial preference of point i to take point m as its cluster center. $r(i,\ m)$ represents m's advantage in competing with other points to become the cluster center of point i. If another point m' is very attractive to point i (i.e., point i extremely recognizes point m' as its own cluster center, therefore $a(i,\ m')$ and $s(i,\ m')$ are larger) and i's initial preference for m is relatively low (i.e., $s(i,\ m)$ is smaller), the responsibility $r(i,\ m)$ will be very low. $a(i,\ m)$ represents the voter's evaluation of the candidate. If other points (e.g., i') all recognize point m as their cluster center (i.e., $r(i',m)$ is relatively large), point i will naturally agree and make

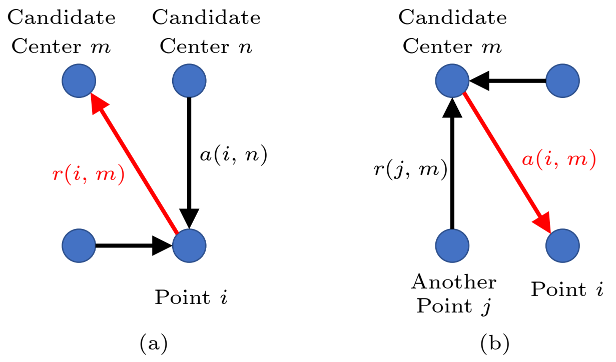


Fig.5. Illustration of sending responsibility and availability[18]. (a) Sending responsibility. (b) Sending availability. For point m, it needs to compete with other points (e.g., n) to become the cluster center of point i. Therefore, $r(i,\ m)$ is affected by $a(i,n)$, i.e., how appropriate that point i recognizes point n as its own cluster center. For point i, when evaluating point m as its cluster center, it needs to refer to the attractiveness of point m to other points (e.g., j), and thus $a(i,\ m)$ is affected by $r(j,\ m)$.

$a(i, m)$ larger. After each iteration, we add $r(i, m)$ and $a(i, m)$ to get the cluster centers. Point m' is the cluster center for point i when it maximizes $(r(i, m)+a(i, m))$. Finally, we decompose the composite commit according to the clustering results.

5 Evaluation Setup

In this section, we present the experimental setup.

5.1 Dataset

There is currently no ground-truth dataset for composite commit detection and untangling. Instead, the common way is to merge a few of existing atomic commits into one composite commit to get the ground truth. In this work, we use the synthetic dataset provided by FLEXEME in our experiments. It is built by first selecting atomic commits in the project based on certain principles (see Herzig *et al.*[2]), and then merging two or three of them into a composite commit (see FLEXEME[13] for more details). In total, there are 1 612 composite commits from nine C# projects collected from GitHub in the original FLEXEME's dataset. The statistics are shown in Table 1.

Table 1. Statistics on the Number of Synthetic Composite Commits in Nine Projects[13]

Project	Two Concerns	Three Concerns	Overall
Commandline	308	32	340
CommonMark	52	0	52
Hangfire	229	87	316
Humanizer	85	4	89
Lean	154	24	178
Nancy	284	67	351
Newtonsoft.Json	84	7	91
Ninject	82	0	82
RestSharp	95	18	113
Overall	1 373	239	1 612

We observe some duplicated commits in the FLEXEME's dataset and perform a de-duplication step. Specifically, two samples are considered as duplicates if the commit graphs of them are identical. After de-duplication, there are a total of 2 062 unique samples, of which 704 are composite commits. The detailed statistics can be found in Table 2.

For COMDET, we use both the atomic commits and the synthesized composite commits. The goal of COMDET is to predict whether the given commit is atomic or composite. Ten-fold cross-validation is conducted to evaluate the effectiveness of COMDET. To further evaluate the effectiveness of COMDET on unseen projects, we also perform a project-based cross-validation, (i.e., the samples from one project are held out for validation and the samples from the other eight projects are used for training in each iteration).

Table 2. Statistics of the Dataset for Evaluating COMDET

Project	Composite	Atomic	Overall
Commandline	120	309	429
CommonMark	19	80	99
Hangfire	140	138	278
Humanizer	40	246	286
Lean	75	161	236
Nancy	167	166	333
Newtonsoft.Json	58	84	142
Ninject	32	97	129
RestSharp	53	77	130
Overall	704	1 358	2 062

For COMUNT, the goal is to split the composite commits into atomic ones. The ground-truth results are available since all the composite commits are synthesized. COMUNT does not require any training data, and we directly apply COMUNT on the commits.

Finally, to evaluate the overall effectiveness of the proposed framework (i.e., COMDET+COMUNT), we randomly divide the dataset in Table 2 into 80% of training data, 10% of validation data, and 10% of test data.

5.2 Baselines

The baselines for COMDET and COMUNT are presented separately. For COMDET, we design a baseline RandomForest on our own as there is no existing work that can detect composite commits based on code diffs. RandomForest adopts random forest[43], a commonly-used machine learning algorithm, to train a classifier to detect composite commits. Specifically, we first use TF-IDF vectors to represent the nodes in a commit graph, and then perform an unweighted average on the vectors of nodes in one graph to obtain graph embedding. Finally, the graph embedding is regarded as the feature of each commit graph and is used to train the random forest algorithm.

We also test the variants of COMDET. We adopt the GGNN model to fuse the structural information and the node features in a commit graph, and design the global context-aware attention to generate a more informative graph embedding. To understand the effects of these components, we build the following variants as comparison methods.

- *COMDET^{-GGNN}*. This variant completely removes the gated graph neural network from COMDET.

That is, the $\boldsymbol{s}_i'$ computed by (1) is used as the embedding of node v_i, and then all node embeddings are aggregated to form a graph embedding using the global context-aware attention;

• *ComDet*$^{-Attn}$. Instead of using the global context-aware attention, this variant simply uses the unweighted average of node embeddings as the final graph embedding. The other modules of this variant are the same as those in ComDet.

To allow a fair comparison between RandomForest and the two variants, the TF-IDF vector is chosen as the initial node vector for the two variants.

For composite commit untangling, we compare ComUnt with the state-of-the-art statement-level commit untangling approach Flexeme[13], and two approaches from Barnett *et al.*[6] and Herzig *et al.*[2] as follows.

• Flexeme builds a δ-NFG for each composite commit, and then calculates the first-order-neighbor subgraph similarity between two nodes. It finally applies the standard agglomerative clustering to obtain atomic commits.

• Barnett *et al.* recognized code entities as nodes and def-use relationships as edges, and then clustered nodes.

• Herzig *et al.* used confidence voters to construct a distance matrix for each composite commit, and then applied a modified multilevel graph partitioning algorithm.

We do not make comparison with SmartCommit[14] mainly because: 1) SmartCommit is designed to require developer participation in the untangling process, whereas ComUnt does not; 2) SmartCommit is implemented for Java projects, whereas ComUnt is implemented for C# projects. It is difficult and unfair to compare SmartCommit and ComUnt directly.

Finally, we mainly compare the combined ComDet and ComUnt approach with the state-of-the-art approach Flexeme.

5.3 Evaluation Metrics

To measure the performance of ComDet, we use the precision, recall, $F1$-score, and accuracy metrics as they are well-known metrics for binary classification, and the composite commit detection problem is essentially a binary classification problem.

For ComUnt, since it is a lazy approach, both accuracy and efficiency are considered. For accuracy, we measure the following metric, which is also taken by the baseline methods:

$$Acc = \frac{\text{number of correctly labeled nodes}}{\text{number of nodes in the graph}},$$

where we compute the fraction of correctly labeled nodes. Acc is also used to assess the performance of the overall approach (ComDet+ComUnt). Each node representing the changed statement in a graph has an original label, which means that it should be untangled into which atomic submission. ComUnt will calculate another label for these nodes. If the two labels of a node are the same, it means that the node is labeled correctly. Acc is just the ratio of the number of correctly labeled nodes to the number of nodes representing the changed statement. There is a case where the ground-truth result is "[01222]" and the prediction result is "[20111]". This is a perfect untangling result as the five code statements are correctly decomposed into three atomic commits. However, a simple evaluation function will yield zero accuracy. Such cases are handled by considering different mappings between the ground-truth cluster IDs and the prediction cluster IDs, and the maximum accuracy is reported. For efficiency, we record the wall-clock time taken by the approaches.

All experiments are run on the same machine, and little interference is introduced when the experiments are run.

5.4 Implementation

Our Approach. In ComDet, the layers and hidden size of the GGNN are set to 8 and 128, respectively, and the dropout rate is set to 0.1. We use *TfidfVectorizer* in scikit-learn[44] with *max_features* = 1 000 to generate a TF-IDF vector for each commit graph node. The pre-trained models BERT[4] and CodeBERT[5] are downloaded from Hugging Face. In the training stage, we adopt the binary cross-entropy loss function and the Adam[45] optimizer with learning rate of 0.000 1. The batch size and maximum training epochs are set to 24 and 300, respectively. We evaluate the model after every epoch and stop training when the loss on the evaluation set no longer drops for 10 epochs. In ComUnt, the maximum num-

[4]https://huggingface.co/bert-base-cased, Nov. 2024.
[5]https://huggingface.co/microsoft/codebert-base, Nov. 2024.

ber of graph convolutions is set to 50, which means that each node can obtain information from a maximum of 50-hop neighbors.

Baselines. We implement RandomForest based on the widely used library scikit-learn[44]. A total of 100 decision trees are used in the forest, and each tree uses the *gini* function to choose the best splitter. For the three baselines of ComUnt, we reuse the implementations provided by Flexeme.

Environment. The experiments are carried out on a Dell computer equipped with an Intel® Core™ i9-10900KF @ 3.70 GHz processor and Ubuntu 18.04.1. ComDet and its variants are trained on one GeForce RTX 3090 with 24 G of VRAM.

6 Evaluation Result

In this section, we present the evaluation results. Specifically, our experiments are designed to answer the following research questions.

• $RQ1$. How effective is the overall framework for composite commit untangling when the input contains both atomic and composite commits?

• $RQ2$. How effective is ComDet in composite commit detection?

• $RQ3$. What are the effects of the GGNN, the global context-aware attention, and the initial node vectors on the performance of ComDet?

• $RQ4$. How accurate can ComUnt untangle a composite commit into atomic commits? Is ComUnt better than the state-of-the-art approaches?

• $RQ5$. As a lazy method, how efficient is ComUnt? Is it faster than the state-of-the-art approaches?

6.1 RQ1: Overall Effectiveness

We first evaluate the performance of the overall framework (ComDet+ComUnt). Note that, without the detection step, the existing commit untangling methods may also split an atomic commit into several parts. To this end, we first evaluate the effectiveness of Flexeme on atomic commits. The results show that Flexeme mis-decomposes 35.2% (478/1 358) of the atomic commits into multiple commits. In contrast, due to the introduced ComDet, our approach can accurately detect atomic commits that do not require further decomposition, and thus has the potential to further improve the final untangling accuracy.

We then show the *Acc* results of Flexeme, ComUnt, and ComDet+ComUnt in Fig.6. In the figure, we intentionally vary the ratios of atomic commits in the test set, so as to show the effectiveness of different methods under different cases. It can be observed that ComDet+ComUnt achieves the best performance regardless of the ratios of atomic commits. Specifically, compared with Flexeme, ComDet+ComUnt achieves a relative improvement ranging from 3.1% to 8.2%, with an average relative improvement of 5.1%. Similarly, the average relative improvement of ComDet+ComUnt is 4.4% compared with ComUnt.

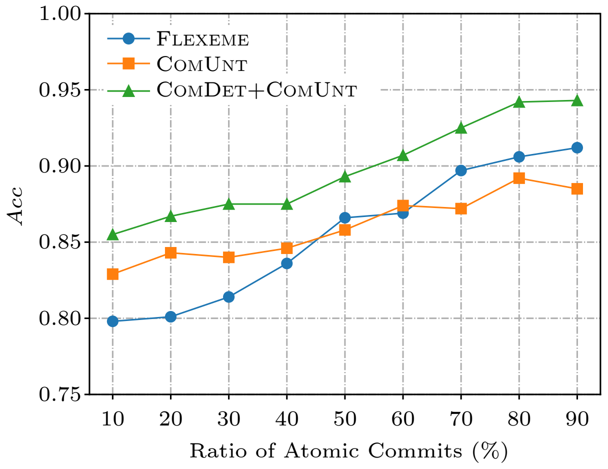


Fig.6. *Acc* of Flexeme, ComUnt, and ComDet+ComUnt under different ratios of atomic commits.

Answer to $RQ1$. The proposed framework performs better than the state-of-the-art approach Flexeme and the ComUnt module alone, when the input contains both atomic and composite commits.

6.2 RQ2: Effectiveness of ComDet

Next, we compare the proposed ComDet with the baseline RandomForest and Table 3 shows the results. ComDet$^{+\mathrm{TI}}$, ComDet$^{+\mathrm{BT}}$, and ComDet$^{+\mathrm{CB}}$ denote ComDet models that use the TF-IDF vector, the BERT vector, and the CodeBERT vector as the initial node vector, respectively. It can be seen that all the three ComDet models generally outperform RandomForest under both random partitioning and by-project partitioning. Specifically, comparing the two approaches, RandomForest and ComDet$^{+\mathrm{TI}}$, which adopt the same type of initial node vector, ComDet$^{+\mathrm{TI}}$ improves recall, $F1$-score, and accuracy by 13.2%, 6.5%, and 0.8%, respectively. The results are similar under the by-project partitioning, and ComDet$^{+\mathrm{TI}}$ achieves 156.0%, 89.7%, and 7.6% improvements in recall, $F1$-score, and accuracy, respec-

Table 3. Effectiveness (%) Results of Composite Commit Detection

Method	Random Partition				By-Project Partition			
	Precision	Recall	$F1$-Score	Accuracy	Precision	Recall	$F1$-Score	Accuracy
RandomForest	**70.0**	46.3	55.7	74.9	**68.0**	19.3	30.1	69.4
COMDET$^{-\text{GGNN}}$	60.5	44.6	51.3	71.1	39.2	18.9	25.5	62.3
COMDET$^{-\text{Attn}}$	64.6	48.2	55.2	73.3	55.6	45.3	49.9	68.8
COMDET$^{+\text{TI}}$	68.3	52.4	59.3	75.5	67.7	49.4	**57.1**	**74.7**
COMDET$^{+\text{BT}}$	68.2	**53.8**	**60.2**	**75.7**	65.6	48.3	55.6	73.7
COMDET$^{+\text{CB}}$	66.9	52.3	58.7	74.9	65.3	**50.3**	56.8	73.9

Note: The best results are in bold. The proposed ComDet generally outperforms the others.

tively.

Note that, all the methods are less effective under by-project partitioning than under random partitioning, suggesting that detecting composite commits in an unseen project is more difficult for supervised learning methods. In addition, with by-project partitioning, the $F1$-score and accuracy of RandomForest decrease by 46.0% and 7.3%, respectively, while the corresponding metrics of ComDet$^{+\text{TI}}$ decrease by only 3.7% and 1.1%, respectively. This suggests that ComDet can extract more cross-project features compared with RandomForest.

*Answer to RQ*2. ComDet performs better than the baseline and achieves up to 89.7% and 7.6% improvements in $F1$-score and accuracy, respectively. Moreover, ComDet can extract more meaningful cross-project features and thus has better generalizability.

6.3 RQ3: Performance Gain of ComDet

To better understand the effects of individual components in ComDet, we design two variants of ComDet and the results are shown in Table 3. As we observe, ComDet models perform better than the two variants, ComDet$^{-\text{GGNN}}$ and ComDet$^{-\text{Attn}}$, which confirms that both the gated graph neural network and the global context-aware attention are helpful in improving the effectiveness of ComDet. Within the two variants, ComDet achieves more improvements compared with ComDet$^{-\text{GGNN}}$ and this indicates that the gated graph neural network is especially important for the composite commit detection task. Additionally, ComDet$^{-\text{GGNN}}$ performs worse than RandomForest under both random and by-project partitions, which again confirms the importance of modeling the commit as a graph.

Moreover, when comparing the three models using different types of initial node vectors, they achieve similar results. This indicates that different types of initial node vectors have limited effects on the performance of ComDet under our experimental setup.

*Answer to RQ*3. Both the gated graph neural network and the global context-aware attention help to improve the effectiveness of ComDet, while different types of initial node vectors have little effects.

6.4 RQ4: Effectiveness of ComUnt

Next, we compare the proposed ComUnt with three baselines, i.e., Barnett *et al.*[6], Herzig *et al.*[2], and FLEXEME[13]. The average *Acc* of different approaches is shown in Table 4. It can be observed that the proposed ComUnt outperforms the other three competitors including the state-of-the-art approach Flexeme. On average, the accuracy of ComUnt is 83%, compared with Flexeme's 77% accuracy, ComUnt achieves a 7.8% relative improvement. Inspecting all projects, we can observe that ComUnt achieves high accuracy rates ranging from 81% to 93%, and it outperforms the three baselines in most cases. ComUnt obtains the largest improvement in the project CommonMark, with a 36.8% relative improvement over Flexeme. The possible reason is that the textual body of code statements in this project is especially important to separate composite commits (e.g., variable names are clearly defined, which in turn benefits commit untangling). In the projects Commandline and Nancy, the accuracy of ComUnt is relatively lower compared with Flexeme. This is due to the simplicity of the composite commits in these projects. For example, the average commit graph size of composite commits in Commandline is 902, which is much smaller than the average commit graph size across all the projects (the average size is 2 431).

Further, Fig.7 shows the accuracy results of Flexeme and ComUnt when the commit graph sizes vary. In order to reflect the changes of accuracy more directly, we first sort the composite commits according to the number of nodes in their commit graphs from

Table 4. Average *Acc* of Composite Commit Untangling Approaches[18]

Approach	# Con.	Project									
		Commandline	CommonMark	Hangfire	Humanizer	Lean	Nancy	Newtonsoft.Json	Ninject	RestSharp	Overall
Barnett	2	0.30	0.26	0.22	0.29	0.24	0.17	0.21	0.19	0.16	0.22
et al. [6]	3	0.25	\	0.20	0.51	0.17	0.11	0.15	\	0.15	0.17
	All	0.30	0.26	0.22	0.30	0.23	0.15	0.21	0.19	0.16	0.22
Herzig	2	0.69	0.66	0.72	0.67	0.71	0.72	0.72	0.64	0.72	0.71
et al. [2]	3	0.53	\	0.59	0.49	0.62	0.58	0.56	\	0.72	0.65
	All	0.67	0.66	0.68	0.66	0.70	0.70	0.71	0.64	0.72	0.69
FLEXEME[13]	2	0.79	0.68	0.81	0.78	0.75	0.78	0.72	0.79	0.78	0.77
	3	**0.89**	\	0.73	0.57	0.78	**0.84**	0.59	\	0.78	0.78
	All	0.80	0.68	0.79	0.77	0.75	0.79	0.71	0.79	0.78	0.77
COMUNT	2	**0.85**	**0.93**	**0.82**	**0.84**	**0.86**	**0.81**	**0.88**	**0.81**	**0.86**	**0.84**
	3	0.80	\	**0.75**	**0.74**	**0.85**	0.82	**0.78**	\	**0.79**	**0.79**
	All	**0.84**	**0.93**	**0.80**	**0.83**	**0.86**	**0.81**	**0.87**	**0.81**	**0.85**	**0.83**

Note: #Con. means the number of concerns, and "2", "3", and "All" refer to the corresponding data of "Two Concerns", "Three Concerns", and "Overall" in Table 1, respectively. The best results are in bold. "\" means that there are no corresponding commits.

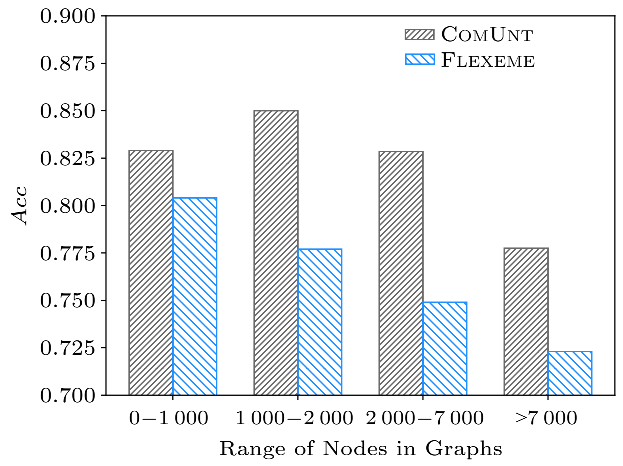


Fig.7. *Acc* vs the number of nodes in commit graphs[18]. The improvement of COMUNT to FLEXEME is even larger for complex commits with relatively large commit graphs.

small to large. We then split them into four groups, with node numbers ranging from 0 to 1 000, 1 000 to 2 000, 2 000 to 7 000, and more than 7 000, respectively. There are 605 385 513, and 109 commits in the four groups, respectively. Then, we calculate the average accuracy in four groups for COMUNT and FLEXEME. It can be observed from the figure that COMUNT's accuracy is higher than FLEXEME's in all groups. Additionally, the improvements become more significant as the commits become more complex. For example, for the commits with more than 2 000 nodes, the average accuracy of COMUNT is 0.83 and the average accuracy of FLEXEME is 0.74, meaning that COMUNT is 12.2% better than FLEXEME. This improvement is more significant compared with the average improvement of 7.8%.

*Answer to RQ*4. COMUNT achieves higher accuracy compared with the existing approaches, and its accuracy ranges from 81% to 93% across different projects. Moreover, the improvement of COMUNT over FLEXEME is even larger for complex commits with relatively large commit graphs, compared with the average case.

6.5 RQ5: Efficiency of COMUNT

Finally, Table 5 shows the average wall-clock time of running each approach in untangling composite commits. COMUNT's average time taken for each project is 4.30–107.58 seconds and the overall average time is 19.35 seconds. This is more than six times faster compared with FLEXEME and more than 1 000 times faster compared with that of Herzig *et al.*[2]. The reason for the efficiency improvement is that COMUNT does not use the time-consuming graph kernel computation as FLEXEME does. Although the approach proposed by Barnett *et al.*[6] is slightly faster, its accuracy is very low. In the project Commandline, COMUNT obtains the largest improvement which is 11 times faster than that of FLEXEME.

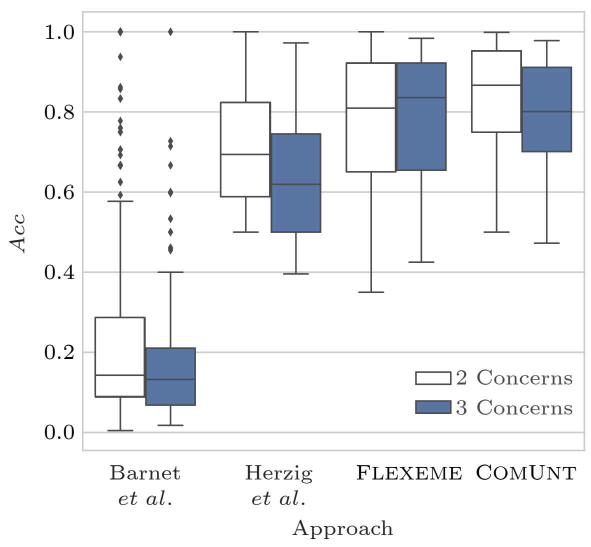
*Answer to RQ*5. COMUNT is very efficient in terms of untangling composite commits, and it is six times faster compared with the state-of-the-art approach FLEXEME.

In Fig.8, we further show the distribution of *Acc* and wall-clock time of the four approaches. It is observed that except for a few outliers, COMUNT's overall accuracy is higher than those of the other three baselines when two concerns are merged into a composite commit. When three concerns are merged, COMUNT's median accuracy is slightly lower than FLEXEME's, but COMUNT is better than FLEXEME in minimum accuracy and lower quartile accuracy, and Co-

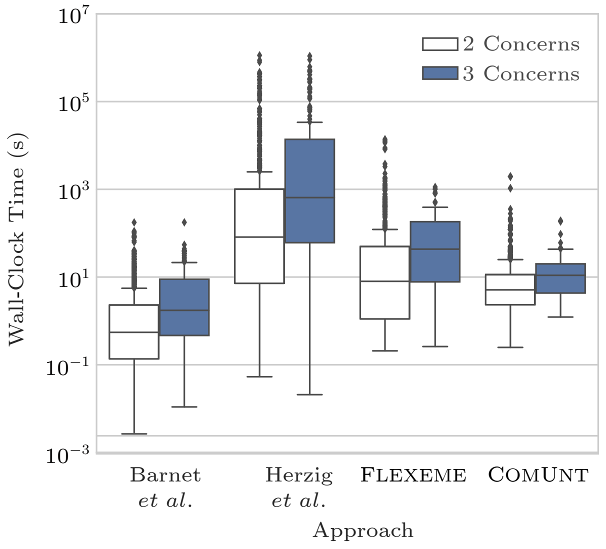
Table 5. Average Wall-Clock Time (s) for Running Composite Commit Untangling Approaches[18]

Approach	# Con.	Project									
		Commandline	CommonMark	Hangfire	Humanizer	Lean	Nancy	Newtonsoft.Json	Ninject	RestSharp	Overall
Barnett	2	4.49	1.83	2.54	0.49	2.07	3.64	8.01	7.16	0.73	3.98
et al. [6]	3	16.25	\	5.58	0.06	1.92	4.99	8.88	\	0.71	6.71
	All	5.60	1.83	3.37	0.47	2.05	3.93	8.08	7.16	0.72	4.41
Herzig	2	671.34	65 697.83	64 155.49	578.18	1 317.21	2 519.79	945.31	397.29	17 950.51	18 272.57
et al. [2]	3	198.43	\	140 044.93	61.26	173.58	1 420.89	510.77	\	23 549.50	55 861.67
	All	616.11	65 697.83	94 288.06	549.98	1 202.84	2 362.80	919.75	397.29	19 397.98	26 437.99
FLEXEME[13]	2	145.63	25.52	50.88	20.47	63.87	83.89	147.48	172.87	794.37	141.85
	3	137.28	\	110.46	2.56	37.72	98.07	27.08	\	304.49	113.64
	All	144.84	25.52	67.28	19.67	60.34	86.60	132.68	172.87	716.33	137.67
COMUNT	2	13.14	4.68	7.92	4.42	11.39	12.69	32.11	15.07	119.14	19.73
	3	10.76	\	16.13	1.69	11.52	15.99	22.80	\	46.61	17.16
	All	12.92	4.68	10.18	4.30	11.40	13.32	31.39	15.07	107.58	19.35

Note: #Con. means the number of concerns, and "2", "3", and "Overall" in the table header refer to the corresponding data of "Two Concerns", "Three Concerns", and "Oerall" in Table 1, respectively. The "\" means that there are no corresponding commits.



(a)



(b)

Fig.8. Boxplot showing the distributions[18]. (a) Distribution of *Acc*. (b) Distribution of wall-clock time.

MUNT's distribution is more concentrated. As for efficiency, COMUNT is faster than FLEXEME when three concerns are merged. When two concerns are merged, COMUNT's median time is roughly equal to FLEXEME's. The overall distribution of COMUNT is also more stable, with a smaller margin between the maximum and minimum time.

7 Threats to Validity

We reuse FLEXEME's dataset in the experiments. This dataset is curated by applying the principles of Herzig *et al.*[2] to select atomic commits from nine project commits, and then tangle these commits into composite commits. Although they have manually checked the atomicity of the selected commits, it is still possible that some of them are not atomic. These non-atomic commits may threaten the results of our experiment. Additionally, the projects studied in the experiments are based on the C# language. Although the commit graph is theoretically common to all programming languages, it still does not indicate that our approach will work better in other languages. Finally, the proposed approach is only evaluated on a synthetic dataset, and how would it perform in real-world scenarios is not clear. This issue can be mitigated by conducting a user study, which is left for future work.

8 Related Work

In this section, we briefly review the related work, including commit classification, composite commit untangling, and graph clustering.

8.1 Commit Classification

Composite commit detection can be seen as a commit classification problem. Prior work on commit classification mainly focuses on predicting the quality[46–48] or purpose[49–54] of commits. For example, Kim *et al.*[46] used Support Vector Machine (SVM) to classify commits as "buggy" or "clean". Hoang *et al.*[47] proposed an end-to-end deep learning framework to predict whether the current commit introduces defects. Zeng *et al.*[48] used a larger dataset to evaluate the approach of Hoang *et al.*[47], analyzed the results in depth and finally proposed a simpler but more effective method. Hiddle *et al.*[49, 50] were interested in the maintenance categories (e.g., Corrective, Perfective, Feature Addition) of commits and adopted many machine learning algorithms for classification. Zhou *et al.*[51] and Zhou *et al.*[52] both used deep learning to identify security patches. Specifically, Zhou *et al.*[51] trained two models based on commit messages and diffs separately, and finally ensembled the two models. Zhou *et al.*[52] first fine-tuned the CodeBERT model and then performed classification based on the fine-tuned model. Meng *et al.*[53] proposed a graph-based commit representation and introduced GNN to classify commits into three categories: bug fixing, functionality addition, and others. Cui *et al.*[54] identified bug fixing commits and further investigated bug fixing activities from the perspective of dependency-level changes.

Our ComDet is different from the above: 1) ComDet is intended to distinguish between composite commits and atomic commits; and 2) ComDet only uses the commit graph, and no other information (e.g., commit messages) is needed.

8.2 Composite Commit Untangling

Composite commits have been of interest to researchers for a long time. Murphy-Hill *et al.*[55] found that developers frequently interleave refactoring with other types of programming activities, e.g., adding a feature or fixing a bug, in their study on refactoring practices. Later, Tao *et al.*[1] first introduced the notion of composite changes (the changes involve a large number of files, span a lot of features or address multiple issues), and highlighted the need for automatically decomposing a composite change. Herzig *et al.*[2, 4] found that up to 20% of bug fixing commits are composite commits and these composite commits can severely affect the effectiveness of bug-prone file prediction models.

Recently, various efforts have been spent on untangling composite commits. For example, Dias *et al.*[8] collected features during development and clustered fine-grained code changes into possible untangled commits. Later approaches are mainly graph-based due to various code dependencies. Barnett *et al.*[6] recognized code entities as nodes and def-use relationships as edges, and then clustered nodes. Guo *et al.*[56] proposed an approach to untangle composite changes and identify a subset of related atomic changes, and then used the atomic changes for the regression testing process. Nguyen *et al.*[21] used natural language processing to analyze the sentences in commit logs and used program analysis to cluster the changes in the change sets to determine if a changed file is for non-fixing. Wang *et al.*[11] decomposed a commit into independent parts by code dependency analysis and tree-based similar-code detection. Flexeme builds the δ-NFG and applies graph kernel based clustering to it. The proposed approach is also based on graphs. Compared with the existing work, our graph additionally considers the semantic information in the textual body of code statements.

8.3 Graph Clustering

Graph clustering algorithms can be divided into flow-based and spectrum-based ones. For example, Flake *et al.*[34] mined communities from social graphs by solving the maximum-flow and minimum-cut problems. Gkantsidis *et al.*[35] performed spectral analysis of the Internet topology at the autonomous system (AS) level and acquired some clusters of ASes. Bo *et al.*[57] used a graph neural network to update node features and then cluster nodes. To fit the commit untangling task, we design a new clustering algorithm by combining flow-based and spectrum-based algorithms. Specifically, we encode both node attributes and the graph structure in the spectrum domain, and then propagate the encoded node features through affinity propagation.

9 Conclusions

This paper presented a novel approach to detect and untangle composite commits. The proposed approach consists of three stages: commit graph construction, composite commit detection, and composite commit untangling. In each stage, it takes into account both the textual body of code statements as well as various dependencies between code statements. Experimental results show that the proposed approach outperforms several existing approaches in both effectiveness and efficiency. In the future, we plan to collect larger datasets to evaluate the effectiveness of the proposed approach. We are also interested in including human feedback during the commit untangling process.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Tao Y, Dang Y, Xie T, Zhang D, Kim S. How do software engineers understand code changes?: An exploratory study in industry. In *Proc. the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2012, Article No. 51. DOI: 10.1145/2393596.2393656.

[2] Herzig K, Just S, Zeller A. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 2016, 21(2): 303–336. DOI: 10.1007/s10664-015-9376-6.

[3] Herbold S, Trautsch A, Ledel B *et al.* A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, 2022, 27(6): Article No. 125. DOI: 10.1007/s10664-021-10083-5.

[4] Herzig K, Zeller A. The impact of tangled code changes. In *Proc. the 10th Working Conference on Mining Software Repositories* (*MSR*), May 2013, pp.121–130. DOI: 10.1109/msr.2013.6624018.

[5] Nguyen H A, Nguyen A T, Nguyen T N. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Proc. the 24th IEEE International Symposium on Software Reliability Engineering* (*ISSRE*), Nov. 2013, pp.138–147. DOI: 10.1109/issre.2013.6698913.

[6] Barnett M, Bird C, Brunet J, Lahiri S K. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. the 37th IEEE/ACM IEEE International Conference on Software Engineering*, May 2015, pp.134–144. DOI: 10.1109/icse.2015.35.

[7] Tao Y, Kim S. Partitioning composite code changes to facilitate code review. In *Proc. the 12th IEEE/ACM Working Conference on Mining Software Repositories*, May 2015, pp.180–190. DOI: 10.1109/msr.2015.24.

[8] Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S. Untangling fine-grained code changes. In *Proc. the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (*SANER*), Mar. 2015, pp.341–350. DOI: 10.1109/saner.2015.7081844.

[9] Kirinuki H, Higo Y, Hotta K, Kusumoto S. Splitting commits via past code changes. In *Proc. the 23rd Asia-Pacific Software Engineering Conference* (*APSEC*), Dec. 2016, pp.129–136. DOI: 10.1109/apsec.2016.028.

[10] Muylaert W, De Roover C. Untangling composite commits using program slicing. In *Proc. the 18th IEEE International Working Conference on Source Code Analysis and Manipulation* (*SCAM*), Sept. 2018, pp.193–202. DOI: 10.1109/SCAM.2018.00030.

[11] Wang M, Lin Z, Zou Y, Xie B. CoRA: Decomposing and describing tangled code changes for reviewer. In *Proc. the 34th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), Nov. 2019, pp.1050–1061. DOI: 10.1109/ase.2019.00101.

[12] Yamashita S, Hayashi S, Saeki M. ChangeBeadsThreader: An interactive environment for tailoring automatically untangled changes. In *Proc. the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering* (*SANER*), Feb. 2020, pp.657–661. DOI: 10.1109/saner48275.2020.9054861.

[13] Pârtachi P P, Dash S K, Allamanis M, Barr E T. Flexeme: Untangling commits using lexical flows. In *Proc. the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2020, pp.63–74. DOI: 10.1145/3368089.3409693.

[14] Shen B, Zhang W, Kästner C, Zhao H, Wei Z, Liang G, Jin Z. SmartCommit: A graph-based interactive assistant for activity-oriented commits. In *Proc. the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug. 2021, pp.379–390. DOI: 10.1145/3468264.3468551.

[15] Hindle A, Barr E T, Su Z, Gabel M, Devanbu P. On the naturalness of software. *Communications of the ACM*, 2016, 59(5):122–131. DOI: 10.1145/2902362.

[16] Zhang X, Liu H, Li Q, Wu X M. Attributed graph clustering via adaptive graph convolution. In *Proc. the 28th International Joint Conference on Artificial Intelligence*, Aug. 2019, pp.4327–4333. DOI: 10.24963/ijcai.2019/601.

[17] Frey B J, Dueck D. Clustering by passing messages between data points. *Science*, 2007, 315(5814): 972–976. DOI: 10.1126/science.1136800.

[18] Chen S Y, Xu S B, Yao Y, Xu F. Untangling composite commits by attributed graph clustering. In *Proc. the 13th Asia-Pacific Symposium on Internetware*, Jun. 2022, pp.117–126. DOI: 10.1145/3545258.3545267.

[19] Nguyen A T, Nguyen T N. Graph-based statistical language model for code. In *Proc. the 37th IEEE/ACM IEEE International Conference on Software Engineering*, May 2015, pp.858–868. DOI: 10.1109/icse.2015.336.

[20] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In *Proc. the 6th International Conference on Learning Representations*, Apr. 30–May 3, 2018.

[21] Nguyen H A, Nguyen T N, Dig D, Nguyen S, Tran H, Hilton M. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering* (*ICSE*), May 2019, pp.819–830. DOI: 10.1109/icse.2019.00089.

[22] Shen B, Zhang W, Zhao H, Liang G, Jin Z, Wang Q. IntelliMerge: A refactoring-aware software merging tech-

nique. *Proceedings of the ACM on Programming Languages*, 2019, 3(OOPSLA): Article No. 170. DOI: 10.1145/3360596.

[23] Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proc. the 33rd International Conference on Neural Information Processing Systems*, Dec. 2019, Article No. 915.

[24] Zhang K, Wang W, Zhang H, Li G, Jin Z. Learning to represent programs with heterogeneous graphs. In *Proc. the 30th IEEE/ACM International Conference on Program Comprehension*, May 2022, pp.378–389. DOI: 10.1145/3524610.3527905.

[25] Kumar K S, Malathi D. A novel method to find time complexity of an algorithm by using control flow graph. In *Proc. the 2017 International Conference on Technical Advancements in Computers and Communications* (*ICTACC*), Apr. 2017, pp.66–68. DOI: 10.1109/ictacc.2017.26.

[26] Kavi K M, Buckles B P, Bhat U N. A formal definition of data flow graph models. *IEEE Transactions on Computers*, 1986, 35(11): 940–948. DOI: 10.1109/tc.1986.1676696.

[27] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987, 9(3): 319–349. DOI: 10.1145/24039.24041.

[28] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In *Proc. the 2014 IEEE Symposium on Security and Privacy*, May 2014, pp.590–604. DOI: 10.1109/sp.2014.44.

[29] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. In *Proc. the 5th International Conference on Learning Representations*, Apr. 2017.

[30] Li Y, Tarlow D, Brockschmidt M, Zemel R S. Gated graph sequence neural networks. In *Proc. the 4th International Conference on Learning Representations*, May 2016.

[31] Xu K, Hu W, Leskovec J, Jegelka S. How powerful are graph neural networks? In *Proc. the 7th International Conference on Learning Representations*, May 2019.

[32] Brandes U, Gaertler M, Wagner D. Experiments on graph clustering algorithms. In *Proc. the 11th Annual European Symposium on Algorithms*, Sept. 2003, pp.568–579. DOI: 10.1007/978-3-540-39658-1_52.

[33] Carrasco J J, Fain D C, Lang K J, Zhukov L. Clustering of bipartite advertiser-keyword graph. In *Proc. the 3rd IEEE International Conference on Data Mining, Workshop on Clustering Large Data Sets*, Nov. 2003.

[34] Flake G W, Lawrence S, Giles C L, Coetzee F M. Self-organization and identification of web communities. *Computer*, 2002, 35(3): 66–70. DOI: 10.1109/2.989932.

[35] Gkantsidis C, Mihail M, Zegura E. Spectral analysis of internet topologies. In *Proc. the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (*IEEE Cat. No. 03CH37428*), Mar. 2003, pp.364–374. DOI: 10.1109/infcom.2003.1208688.

[36] Mihail M, Gkantsidis C, Saberi A. On the semantics of internet topologies. Technical report, Georgia Institute of Technology, 2002. https://repository.gatech.edu/entities/publication/88498271-8732-47f2-898e-f701ca23fcba/full, Nov. 2022.

[37] Tian F, Gao B, Cui Q, Chen E, Liu T Y. Learning deep representations for graph clustering. In *Proc. the 38th AAAI Conference on Artificial Intelligence*, Jul. 2014, pp.1293–1299. DOI: 10.1609/aaai.v28i1.8916.

[38] Wang C, Pan S, Hu R, Long G, Jiang J, Zhang C. Attributed graph clustering: A deep attentional embedding approach. In *Proc. the 28th International Joint Conference on Artificial Intelligence*, Aug. 2019, pp.3670–3676. DOI: 10.24963/ijcai.2019/509.

[39] Shervashidze N, Schweitzer P, van Leeuwen E J, Mehlhorn K, Borgwardt K M. Weisfeiler-Lehman graph kernels. *The Journal of Machine Learning Research*, 2011, 12: 2539–2561. DOI: 10.5555/1953048.2078187.

[40] Dash S K, Allamanis M, Barr E T. RefiNym: Using names to refine types. In *Proc. the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2018, pp.107–117. DOI: 10.1145/3236024.3236042.

[41] Devlin J, Chang M W, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proc. the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1* (*Long and Short Papers*), Jun. 2019, pp.4171–4186. DOI: 10.18653/v1/N19-1423.

[42] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In *Proc. the Findings of the Association for Computational Linguistics: EMNLP 2020*, Nov. 2020, pp.1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[43] Liaw A, Wiener M. Classification and regression by randomforest. *R News*, 2002, 2(3): 18–22.

[44] Pedregosa F, Varoquaux G, Gramfort A *et al*. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 2011, 12: 2825–2830.

[45] Kingma D P, Ba L J. Adam: A method for stochastic optimization. In *Proc. the 3rd International Conference on Learning Representations* (*ICLR*), May 2015, Article No. 13.

[46] Kim S, Whitehead E J, Zhang Y. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 2008, 34(2): 181–196. DOI: 10.1109/tse.2007.70773.

[47] Hoang T, Dam H K, Kamei Y, Lo D, Ubayashi N. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In *Proc. the 16th IEEE/ACM International Conference on Mining Software Repositories* (*MSR*), 2019, pp.34–45. DOI: 10.1109/msr.2019.00016.

[48] Zeng Z, Zhang Y, Zhang H, Zhang L. Deep just-in-time defect prediction: How far are we? In *Proc. the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2021, pp.427–438. DOI: 10.1145/3460319.3464819.

[49] Hindle A, German D M, Holt R. What do large commits tell us?: A taxonomical study of large commits. In *Proc.*

the 2008 International Working Conference on Mining Software Repositories, May 2008, pp.99–108. DOI: 10.1145/1370750.1370773.

[50] Hindle A, German D M, Godfrey M W, Holt R C. Automatic classication of large changes into maintenance categories. In *Proc. the 17th IEEE International Conference on Program Comprehension*, May 2009, pp.30–39. DOI: 10.1109/icpc.2009.5090025.

[51] Zhou Y, Siow J K, Wang C, Liu S, Liu Y. SPI: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022, 31(1): Article No. 13. DOI: 10.1145/3468854.

[52] Zhou J, Pacheco M, Wan Z, Xia X, Lo D, Wang Y, Hassan A E. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *Proc. the 36th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), Nov. 2021, pp.705–716. DOI: 10.1109/ase51524.2021.9678720.

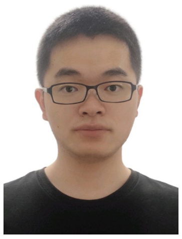
[53] Meng N, Jiang Z, Zhong H. Classifying code commits with convolutional neural networks. In *Proc. the 2021 International Joint Conference on Neural Networks* (*IJCNN*), Jul. 2021, pp.1–8. DOI: 10.1109/ijcnn52387.2021.9533534.

[54] Cui D, Fan L, Chen S, Cai Y, Zheng Q, Liu Y, Liu T. Towards characterizing bug fixes through dependency-level changes in apache java open source projects. *Science China Information Sciences*, 2022, 65(7): 172101. DOI: 10.1007/s11432-020-3317-2.

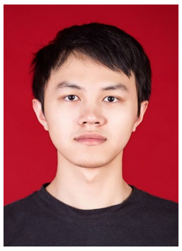
[55] Murphy-Hill E, Parnin C, Black A P. How we refactor, and how we know it. In *Proc. the 31st IEEE International Conference Software Engineering*, May 2009, pp.287–297. DOI: 10.1109/icse.2009.5070529.

[56] Guo B, Kwon Y W, Song M. Decomposing composite changes for code review and regression test selection in evolving software. *Journal of Computer Science and Technology*, 2019, 34(2): 416–436. DOI: 10.1007/s11390-019-1917-9.

[57] Bo D, Wang X, Shi C, Zhu M, Lu E, Cui P. Structural deep clustering network. In *Proc. the Web Conference 2020*, Apr. 2020, pp.1400–1410. DOI: 10.1145/3366423.3380214.



Sheng-Bin Xu currently is a Ph.D. candidate at the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing. He received his B.S. degree in computer science from Nanjing University, Nanjing, in 2017. His major research interests include code change representation and deep learning for software engineering.



Si-Yu Chen currently is a master student at the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing. He received his B.S. degree in software engineering from the University of Electronic Science and Technology of China, Chengdu, in 2020. His major research interest includes deep learning for software engineering especially about code changes.

Yuan Yao is an associate professor at the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing. His current research interests include machine learning for software analysis with special focus on automation and quality aspects, quality assurance for machine learning with special focus on robustness and reliability aspects, networked data mining and its applications to social media analytics, and recommender systems.

Feng Xu is a professor at the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing. He received his B.S. and M.S. degrees in computer science from Hohai University, Nanjing, in 1997 and 2000, respectively. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2003. His research interests include software defect localization, data mining, recommender systems, and trust management.