Software Defect Prediction via Positional Hierarchical Attention Network

Xinyan Yi[1], Hao Xu[1], Lu Lu[1,3*], Quanyi Zou[2,3], Zhanyu Yang[1]
[1]School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
[2]School of Journalism and Communication, South China University of Technology, Guangzhou, China
[3]Pazhou Laboratory, Guangzhou, China
*Corresponding author email: lul@scut.edu.cn

***Abstract*—Software Defect Prediction (SDP) aims to identify defect-prone modules in advance to ensure software quality. In SDP research based on deep learning, the mainstream approach is to extract deep semantic features from an Abstract Syntax Tree (AST). Theoretically, the AST as a bi-dimensional structure encloses information at the node level, fragment level, and entire tree level. However, most existing research serializes the whole AST without considering the expression at different granularities. To address this limitation, we introduce a positional hierarchical attention network (PHAN) that acquires semantic features by simultaneously considering contexts between nodes and paths. Specifically, our model incorporates attention mechanisms to capture information of varying importance at separate hierarchies, and relative position representations to distinguish the contributions of different paths. Experimental results demonstrate that PHAN significantly outperforms existing baseline methods.**

***Keywords*—software defect prediction; abstract syntax tree; hierarchical attention; deep semantic feature**

I. Introduction

As the scale and complexity of software continue to boost, software defects pose greater challenges to software quality. SDP is the technique that leverages historical data to identify defect-prone software modules in advance, enabling the enhancement of software quality and testing efficiency.

Early research in SDP using traditional hand-crafted features is limited to capturing semantic information. With the emergence of deep learning, some studies applied natural language processing methods to encode codes as general texts [1]. However, symbols like arithmetic operators may involve useless ambiguous information. AST as a representation of the code's syntax structure, inherently encapsulating the syntax structure and semantic information, has shown its promising performance in numerous code-related tasks [2]. In SDP tasks, AST is commonly presented through traversal [3], tree-based networks [4], and graph-based networks [5].

However, previous research methods on semantic encoding in SDP still have limitations. Firstly, whether AST is flattened as a text sequence or maintains its original structure, they encode the AST as a whole, ignoring the information at a moderate granularity level. Secondly, although some studies in SDP have taken a hierarchical structure into account, such as decomposing code into tokens and lines levels [1], and splitting AST into nodes and subtrees levels [6], there is a lack of encoding at the path granularity in SDP. Furthermore, existing AST presentation by mining paths [7] does not consider the positional information between paths, while the positional difference may indicate the existence of defects.

In this paper, we propose a positional hierarchical attention network for SDP. The main contributions are as follows:

1) Design a method for learning AST representation based on hierarchical path mining and tree reconstruction.

2) Introduce relative position encoding at the path level to alleviate the lack of long-distance path dependencies in SDP.

II. Related Works

A. Code Representations in Defect Prediction

Majd et al. represented code in the forms of tokens, lines, and statements, which focus on the code's high level [8]. Wang et al. transformed the source code into AST and extracted vectors through pre-order traversal of the AST [9]. Phan et al. argued that the Control Flow Graph can better represent code information in SDP [10]. Tian et al. extracted code slices and converted them into System Dependency Graphs for control flow and data flow analysis [11]. Chen et al. transformed each character into a pixel, visualizing the programs as images [12].

B. Deep Learning Approaches for Defect Prediction

Wang et al. suggested using Deep Belief Networks (DBNs) and source code changes to capture semantic feature representations [9]. Then, Li et al. combined semantic features extracted by Convolutional Neural Networks (CNN) with hand-crafted features, which improved prediction performance by exploiting their superior ability to mine local features [13]. Dam et al. leveraged a tree-structured Long Short-term Memory (LSTM) network to automatically mine long context dependency in code [4]. Uddin et al. employed Bi-LSTM to learn contextual information from the token vectors embedded through the pre-trained model BERT [14].

III. Method

In this section, the framework of the proposed model is illustrated in Fig. 1. During the preprocessing stage, JAVA files are transformed into ASTs, simultaneously tracking the paths of trees. In the encoding stage, semantic features are extracted hierarchically at both the node and path levels. Lastly, the resulting AST encoding is subsequently fed into a classifier.

DOI reference number: 10.18293/SEKE23-119

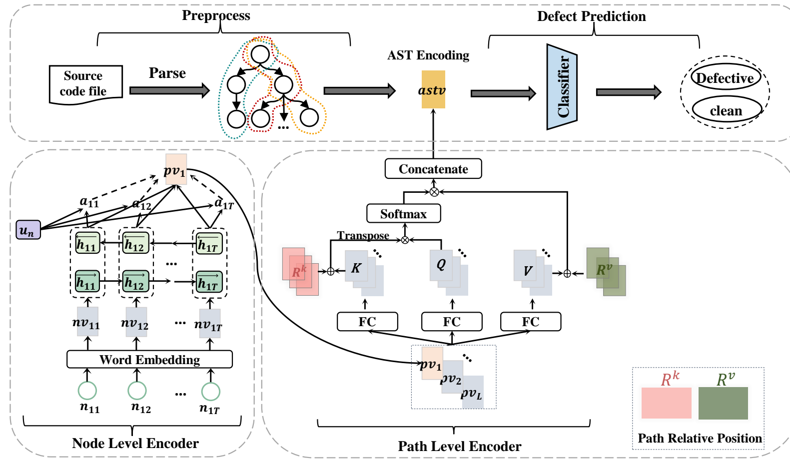


Fig. 1. The framework of PHAN.

A. Source Code File Preprocessing

Prior to encoding, toolkit javalang[1] parses the Java code into an AST. Then, all root-to-leaf paths are extracted by a traversal, and nodes are initialized as vectors through an embedding layer:

$$nv_{it} = Embedding(n_{it}), t \in [1, T], i \in [1, L], \tag{1}$$

where nv_{it} denotes an initial vector of the t-th node in the i-th path, T is the number of nodes in the longest path, L is the number of paths in the biggest AST.

B. Node Level Encoder

At the node level, the additive attention [15] is adopted.

Node addictive attention. Given the set of preliminary vector representation $N = [nv_{i1},\ nv_{i2}, ..., nv_{iT}]$, the annotations of nodes are firstly obtained through a bidirectional GRU (Bi-GRU) [16] layer. We get the node annotation $h_{i,t}$ incorporating node contextual information by concatenating the output of forward direction $\overrightarrow{h_{i,t}}$ and the backward direction $\overleftarrow{h_{i,t}}$:

$$\overrightarrow{h_{i,t}} = \overrightarrow{GRU}(nv_{i,t}), t \in [1, T], \tag{2}$$

$$\overleftarrow{h_{i,t}} = \overleftarrow{GRU}(nv_{i,t}), t \in [T, 1], \tag{3}$$

$$h_{i,t} = [\overrightarrow{h_{i,t}}, \overleftarrow{h_{i,t}}]. \tag{4}$$

Bi-GRU has the potential to learn positional information, making additional position information unnecessary. Subsequently, the node annotation $h_{i,t}$ is directly fed into a single-layer perceptron to obtain its nonlinear representation $u_{i,t}$:

$$u_{i,t} = \tanh\left(W_n h_{i,t} + b_n\right). \tag{5}$$

Then the similarity of a node-level contextual vector u_n with each hidden representation $u_{i,t}$ is computed as importance metrics and normalized through a softmax function to obtain an importance weight $\alpha^N_{i,t}$:

$$\alpha^N_{i,t} = softmax\left(u_{i,t}{}^\top u_n\right), \tag{6}$$

where u_n is initialed as a trainable parameter.

[1]https://github.com/c2nes/javalang

Path encoding. All the critical node information is aggregated by taking a weighted sum of the hidden states and attention weights, resulting in the entire path encoding pv_i:

$$pv_i = \sum_t^T \alpha^N_{i,t} h_{i,t}. \tag{7}$$

C. Path Level Encoder

Hierarchical Attention Network (HAN) [17] leverages a shared RNN encoder and attention mechanism for word-level and sentence-level. However, in SDP, the number of paths L is significantly larger than the number of nodes T, resulting in a long sequence where Bi-GRU is limited to capturing long-range dependencies due to the gradual decay. Thus we propose a positional self-attention mechanism at the path level.

Path positional self-attention. Given the set of path vector $P = [pv_1, ..., pv_L]$, self-attention[4] can be defined as:

$$\begin{gathered} SelfAtt(Q, K, V) = V softmax\left(\frac{K^\top Q}{\sqrt{d}}\right) \\ K = W_K P, Q = W_Q P, V = W_V P, \end{gathered} \tag{8}$$

where d denotes the dimension of each path vector pv, W_K, W_Q and W_V are parameter matrices.

Self-attention does not consider the positions which often carry semantic meaning, so we introduce relative position embedding [18]. The self-attention score between the i-th and j-th path is measured by the scaled dot product of $q_i \in Q$ and $k_j \in K$. In absolute positional encoding, the positional embedding pos is added straightforwardly to the input vector, then the self-attention weight $\alpha^P_{i,j}$ is calculated as:

$$\begin{aligned} \alpha^P_{i,j} &= softmax\left(\frac{k_j{}^\top q_i}{\sqrt{d}}\right) \\ &= softmax\left(\frac{(W_K pv_j + W_K pos_j)^\top (W_Q pv_i + W_Q pos_i)}{\sqrt{d}}\right). \end{aligned} \tag{9}$$

In relative positional encoding, the position of each path is not explicitly represented, instead, the relative distance between the current position and the position being attended to is taken into account. We eliminate the $W_Q pos_i$ and replace

TABLE I. STATISTICAL OVERVIEW OF DATASETS

Project	Versions	Files	Defective(%)
Ant	1.5, 1.6	293, 351	10.9, 26.2
Jedit	4.0, 4.1	306, 312	24.5, 25.3
Log4j	1.0, 1.1	135, 109	25.2, 33.9
Lucene	2.2, 2.4	247, 340	58.3, 59.7
Poi	2.5.1, 3.0	385, 442	64.4, 63.6
Synapse	1.1, 1.2	222, 256	27.0, 33.6
Velocity	1.5, 1.6.1	214, 229	66.4, 34.1
Xalan	2.5, 2.6	803, 885	48.3, 46.4
Xerces	1.3, 1.4.4	453, 588	15.2, 74.3

the $W_K pos_j$ with a binary positional vector $R^K_{i,j}$, then the attention weights after transformation is as follows:

$$\alpha^P_{i,j} = softmax\left(\frac{(W_K pv_j + R^K_{i,j})^\mathsf{T} W_Q pv_i}{\sqrt{d}}\right). \quad (10)$$

Analogously, the $v_j \in V$ is generated by adding positional encoding to the input vectors. We also substitute the $W_V pos_j$ with a binary position vector $R^V_{i,j}$ and weighted by multiplication with the attention score $\alpha^P_{i,j}$ to obtain the output o_i:

$$\begin{aligned} o_i &= \sum_j \alpha^P_{i,j} v_j = \sum_j \alpha^P_{i,j}(W_V pv_j + W_V pos_j) \\ &= \sum_j \alpha^P_{i,j}(W_V pv_j + R^V_{i,j}). \end{aligned} \quad (11)$$

AST encoding. All paths are aggregated by taking a weighted sum of the outputs $O = [o_1, ..., o_L]$, resulting in the AST encoding $astv$:

$$astv = OW_{ast} = \sum_i^L \beta_i o_i \ , \ W_{ast} = [\beta_1, ...\beta_L]^\mathsf{T}, \quad (12)$$

where $W_{ast} \in \mathbb{R}^{L\times 1}$ denotes a trainable parameter matrix.

D. Software Defect Prediction via PHAN

This AST vector $astv$ is utilized as the semantic feature and fed into a Logistic Regression (LR) for defect prediction.

IV. EXPERIMENTAL SETTING

A. Experimental Datasets

The dataset employed is the classic SDP repository PROMISE [19], detailed information is presented in Table I. This paper chooses the old version source code used for training and the new one for testing.

B. Evaluation Metrics

Higher F-measure values indicate more robustness. MCC provides a better evaluation when dealing with imbalanced distributions. AUC represented as an area under the curve can avoid experimental errors. Specifically,

$$Precision = \frac{tp}{tp + fp} \ , \ Recall = \frac{tp}{tp + fn}, \quad (13)$$

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall}, \quad (14)$$

$$MCC = \frac{tp * tn - fp * fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}, \quad (15)$$

where tp, fp, tn, and fn refer to True Positive, False Positive, True Negative, and False Negative, respectively.

C. Comparison Methods

PHAN is compared with the following methods:

- LR: A traditional classification method utilized to predict defect-proneness based on hand-craft defect features.
- DBN-WP [9]: A standard DBN model extracting semantic features from AST for within-project defect prediction.
- DP-CNN [13]: An enhanced CNN method based on hand-craft features and deep semantic features for SDP.
- DP-LSTM [3]: A Bi-LSTM Model extracting semantic representations for SDP.
- HAN [17]: Original hierarchical attention network at node and path level for SDP.

V. EXPERIMENTAL RESULTS

This section aims to show comparisons of the proposed method with baselines and analyze the experimental results.

In this study, we investigate the effectiveness of the PHAN compared to a traditional model LR based on hand-crafted features and baseline models based on deep learning covering DBN, CNN, and LSTM. Table II presents that the proposed method outperforms other baseline models in 6 out of 9 F-measure values, 8 out of 9 AUC values, and 7 out of 9 MCC values. Experimental results indicate that PHAN has achieved superior performance compared with a traditional method and several advanced AST-based methods in multiple perspectives. Furthermore, we compare the performance of the proposed PHAN and HAN on SDP tasks. Table II shows that PHAN exceeds HAN across 7 projects. While HAN achieves comparable performance to PHAN on most projects, PHAN performs particularly well on the Xerces project, compensating for the poor performance of HAN on this project.

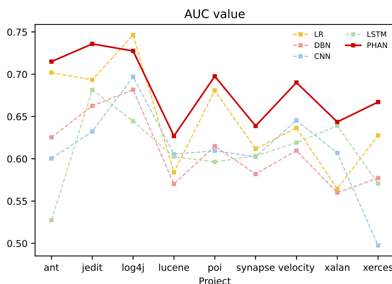
The line chart in Fig. 2 presents a visual comparison of the proposed method and the baseline models, illustrating that PHAN outperforms other models in most cases, particularly in the Ant and Xerces projects. As revealed by the average defect rate of each project shown in Table I, the Ant and Xerces projects of the old version have severe data imbalance problems, posing challenges for learning valuable information from the training set. Notably, all deep learning models exhibit significantly worse performance on the two projects, while only the traditional machine learning model LR achieves similar performance to PHAN. These results highlight that representing ASTs at a finer-grained level and distinguishing the contribution of different paths' positions can learn more effectively from imbalanced data distributions.

VI. CONCLUSION

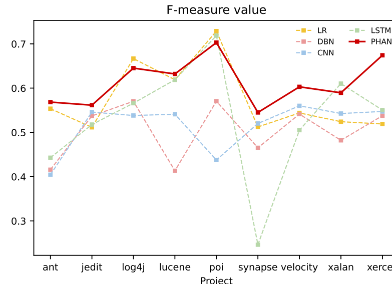
This study presents PHAN, a positional hierarchical attention network model designed to extract code semantic features from ASTs. The PHAN encodes at a finer-grained path level, taking into account the contributions of different nodes and paths to SDP tasks. This approach is empirically compared with traditional methods, state-of-the-art deep learning methods, and original HAN across nine Java projects. Our findings demonstrate that the proposed model achieves the best performance across F-measure, AUC, and MCC values.

TABLE II. F-MEASURE, AUC, MCC COMPARISON OF DIFFERENT MODELS

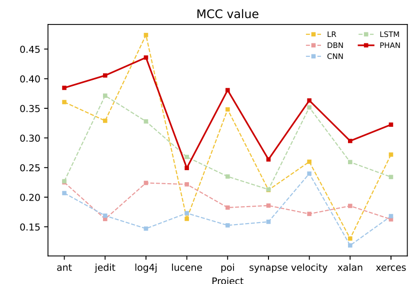
Method	Metric	Project									Average
		Ant	Jedit	Log4j	Lucene	Poi	Synapse	Velocity	Xalan	Xerces	
LR	F-measure	0.553	0.511	**0.667**	0.619	**0.729**	0.512	0.544	0.524	0.519	0.575
	AUC	0.702	0.693	**0.746**	0.584	0.681	0.612	0.636	0.565	0.628	0.650
	MCC	0.361	0.329	**0.474**	0.163	0.348	0.212	0.260	0.130	0.272	0.283
DBN-WP	F-measure	0.416	0.537	0.570	0.413	0.570	0.465	0.541	0.482	0.538	0.504
	AUC	0.625	0.663	0.682	0.570	0.615	0.582	0.610	0.560	0.577	0.609
	MCC	0.225	0.163	0.224	0.222	0.182	0.186	0.172	0.185	0.163	0.191
DP-CNN	F-measure	0.404	0.546	0.538	0.541	0.438	0.520	0.560	0.543	0.547	0.515
	AUC	0.600	0.632	0.697	0.605	0.609	0.602	0.645	0.607	0.498	0.611
	MCC	0.207	0.169	0.147	0.173	0.152	0.158	0.240	0.119	0.168	0.170
DP-LSTM	F-measure	0.443	0.518	0.566	0.619	0.719	0.246	0.505	**0.610**	0.550	0.531
	AUC	0.527	0.681	0.644	0.602	0.516	0.603	0.619	0.639	0.571	0.600
	MCC	0.227	0.371	0.328	**0.268**	0.235	0.213	0.352	0.259	0.234	0.276
HAN	F-measure	0.527	0.553	0.632	**0.652**	0.713	**0.547**	0.596	0.595	0.474	0.588
	AUC	0.679	0.724	0.716	**0.628**	0.684	**0.648**	0.680	0.641	0.604	0.667
	MCC	0.325	0.395	0.416	0.250	0.354	**0.285**	0.342	0.286	0.228	0.320
PHAN	F-measure	**0.568**	**0.561**	0.645	0.632	0.703	0.545	**0.603**	0.589	**0.674**	**0.613**
	AUC	**0.715**	**0.736**	0.727	0.627	**0.698**	0.639	**0.690**	**0.644**	**0.667**	**0.682**
	MCC	**0.385**	**0.405**	0.436	0.249	**0.381**	0.264	**0.363**	**0.295**	**0.322**	**0.344**



(a) AUC values comparison.



(b) F-measure values comparison.



(c) MCC values comparison.

Fig. 2. The results of diverse metrics among baseline models and PHAN.

VII. ACKNOWLEDGMENT

This work was supported by the second batch of cultivation projects of Pazhou Laboratory in 2022, No. PZL2022KF0008.

REFERENCES

[1] C. Pornprasit and C. Tantithamthavorn, "Deeplinedp: Towards a deep learning approach for line-level defect prediction," *IEEE Transactions on Software Engineering*, 2022.

[2] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo, "Ast-trans: code summarization with efficient tree-structured attention," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 150–162, 2022.

[3] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via lstm," *IET Software*, vol. 14, no. 4, pp. 443–450, 2020.

[4] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *arXiv preprint arXiv:1802.00921*, 2018.

[5] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 613–625, 2020.

[6] Y. Zhoua, L. Lua, Q. Zoub, and C. Lic, "Two-stage ast encoding for software defect prediction," in *2022 34th International Conference on Software Engineering & Knowledge Engineering*, pp. 196–199, 2022.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[8] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "Sldeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Systems with Applications*, vol. 147, p. 113156, 2020.

[9] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 297–308, 2016.

[10] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 45–52, IEEE, 2017.

[11] J. Tian and Y. Tian, "A model based on program slice and deep learning for software defect prediction," in *2020 29th International Conference on Computer Communications and Networks*, pp. 1–6, IEEE, 2020.

[12] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov, "Software visualization and deep transfer learning for effective software defect prediction," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 578–589, 2020.

[13] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security*, pp. 318–328, IEEE, 2017.

[14] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, and I. Zada, "Software defect prediction employing bilstm and bert-based semantic feature," *Soft Computing*, vol. 26, no. 16, pp. 7877–7891, 2022.

[15] J. K. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[16] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[17] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1480–1489, 2016.

[18] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," *arXiv preprint arXiv:1803.02155*, 2018.

[19] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pp. 1–10, 2010.