Exploration on Code Representation Techniques in Software Engineering

Charith Kutikuppala[1]

***Abstract*—In the domain of software engineering, the rapid advancement of development techniques necessitates robust and accurate methods for vulnerability detection. Traditional code representation techniques, such as treating queries and code snippets as plain text, are increasingly inadequate for complex analysis tasks. This report investigates advanced deep learning (DL) methods for code representation, including tokens, data flow graphs, and semantic representations, with a particular focus on Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). These methods are compared with pre-trained models, which, despite their computational expense, offer optimal performance. However, this study emphasizes the practical advantages of DL methods like ASTs and CFGs. These techniques provide structured and detailed code representations, enhancing software quality assurance and defect prediction. While the exploration of traversal paths in ASTs and CFGs is ongoing, this report provides a comparative evaluation of their effectiveness relative to pre-trained models, highlighting their pros and cons in software engineering applications.**

I. Introduction

In the field of software engineering, the rapid evolution of development methodologies necessitates innovative and effective techniques for ensuring software quality and detecting vulnerabilities. Traditional approaches to code representation, which often involve treating code snippets and queries as plain text, have proven insufficient for complex analytical tasks. These methods fall short in capturing the intricate structural and semantic nuances of code, leading to less effective vulnerability detection and quality assurance.

The advent of deep learning (DL) has introduced advanced methods for code representation that promise to address these shortcomings. Techniques such as token embeddings, data flow graphs, and semantic representations have significantly improved the ability to analyze and understand code. Among these, Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) stand out for their structured representation of code. ASTs represent the syntactic structure of code, while CFGs capture the control flow, both of which are crucial for various software engineering tasks.

This report aims to compare these DL methods, particularly ASTs and CFGs, with pre-trained models like Codex and CodeBERT. Despite the superior performance of pre-trained models, their high computational costs pose practical limitations. Conversely, DL methods like ASTs and CFGs offer a more balanced approach, combining efficiency with detailed code representation.

A. Problem Statement

The key problem addressed in this report is identifying the most efficient code representation technique for specific software engineering domains, such as defect prediction and code smell detection. The study seeks to determine whether ASTs, CFGs, or other DL-based representations provide the best balance of performance and computational efficiency compared to pre-trained models.

B. Research Questions

This study is guided by the following research questions:

1) **Efficiency of Code Representation Techniques:** Which code representation technique (ASTs, CFGs, or PDGs) proves to be the most efficient for specific software engineering domains like defect prediction and code smell detection?
2) **Comparison with Pre-trained Models:** How do deep learning methods like ASTs and CFGs compare with pre-trained models in terms of performance and computational cost for these applications?

Addressing these questions is crucial as it directly impacts the development of more efficient and effective tools for software quality assurance. Accurate code representation techniques can lead to better defect prediction and code smell detection, ultimately enhancing the reliability and maintainability of software systems. Given the critical role of software in various industries, improvements in these areas can lead to significant advancements in both the development and operational phases of software engineering.

C. Proposed Approach

To address the research questions, this report will undertake a comparative evaluation of ASTs and CFGs relative to pre-trained models, emphasizing the practical advantages of DL methods. The study will involve:

- Reviewing existing literature on traditional and DL-based code representation techniques.
- Conducting experiments to compare the performance of ASTs, CFGs, and pre-trained models on datasets related to defect prediction and code smell detection.
- Analyzing the results to determine the most efficient code representation technique for specific software engineering tasks.

II. Literature Survey

The literature survey encompasses a detailed review of significant works in the domain of software defect prediction, focusing on the comparison of deep learning-based methods and traditional code representation techniques. The following studies are particularly relevant to our research:

Abdu et al. [1] explored deep learning techniques for software defect prediction, leveraging contextual syntax and semantic graphs to enhance predictive accuracy. Their findings indicated that Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) provide substantial improvements in capturing code semantics compared to traditional text-based representations.

Another study by Liu et al. [2] delved into the use of deep learning for source code analysis, demonstrating the effectiveness of ASTs and CFGs in identifying software defects. They emphasized the importance of structured code representations in improving defect detection capabilities.

Wang et al. [3] conducted a comprehensive survey on machine learning techniques for source code analysis, highlighting the advantages of deep learning models over traditional approaches. Their work underscored the potential of ASTs and CFGs in addressing complex software engineering challenges.

These studies collectively underscore the significance of adopting advanced code representation techniques, such as ASTs and CFGs, in enhancing software defect prediction and quality assurance.

III. Methodology

A. PROMISE Dataset

The PROMISE dataset is widely used in the field of software defect prediction. It includes various open-source Java projects with detailed information about each project's versions, average source files, and buggy rates. For this study, we focus on the "ant" project from the dataset, which is a Java-based build tool. The table below provides an overview of the PROMISE dataset used in our experiments.

TABLE I
PROMISE Dataset Description

Project	Description	Versions	Avg Source Files	Avg Buggy Rate (%)
ant	Java-based build code files.	1.5,1.6,1.7	464	22
jedit	A text editor built for programmers.	3.2, 4.0,4.1	297	28.1
camel	Enterprise integration framework.	1.2,1.4, 1.6	815	24.8
log4j	A Java-based logging library.	1.0,1.1	121.8	30.3
xalan	A Java library for processing XML files.	2.4,2.5,2.6	782	36.8
synapse	Adapters for transmitting data.	1.1,1.2	239	30.5
lucene	An open-source text search library.	2.0,2.2,2.4	261	56.3
poi	Java library for accessing Microsoft files.	1.5,2.5,3.0	354.7	63
xerces	XML parser.	1.2,1.3	447.2	15.7

Fig. 1. PROMISE Dataset Description

B. Approach to Generating ASTs and CFGs

Our methodology involves parsing Java source code files to generate Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). These representations are crucial for analyzing the structural and control flow aspects of the code. The following subsections detail the algorithms used for generating these representations.

1) Generating ASTs: We parse AST vectors from Java source files using the Python library 'javalang'. The process involves reading Java program files, extracting the AST nodes, and converting them into vector representations. The following algorithm describes this process:

Algorithm 1 Parsing ASTs' Vectors from Java Source Files

Require: Java Program (P)
Ensure: AST vectors $V = \{v_1, v_2, ..., v_n\}$
Read P
$TREENODES = JAVALANG.PARSE.PARSE(P)$
for N in TREENODES **do**
  if node in N **then**
    Add node into v_i
  end if
  Add v_i into V
end for
return V

2) Generating CFGs: Control Flow Graphs (CFGs) are generated by analyzing the Java source code to identify the control flow statements and constructing the corresponding graph. The algorithm for extracting CFG nodes and edges is described below:

Algorithm 2 Extracting the CFG Nodes and Edges from Java Source Files

Require: Java Program (P)
Ensure: Nodes (N) and Edges (E)
Read P
while find LEADERS **do**
  if preL = nextL **then**
    Add N
  end if
  while find CONTROL **do**
    Add E
  end while
  DrawGraph $G(N, E)$
end while

3) Generating DDGs: Data Dependence Graphs (DDGs) provide additional layers of semantic information by capturing the data dependencies within the code. The generation of DDGs involves analyzing the data flow to identify dependencies between different parts of the code.

C. Illustrative Example

The following figure illustrates different representations of the same Java code, including its AST, CFG, and DDG.

By combining these representations, we aim to leverage their complementary strengths to improve software defect prediction.

IV. Results and Discussions

In this section, we present the results of our analysis on the 'ant' project from the PROMISE dataset using Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). Our primary goal was to evaluate the effectiveness of these code representation techniques in predicting software defects.

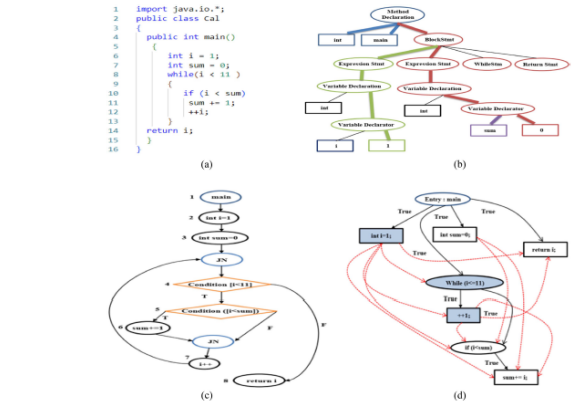

Fig. 2. Example of different representations of source code. (a) Java code example. (b) AST. (c) CFG. (d) DDG.

A. Performance Metrics

The performance of ASTs, CFGs, and pre-trained models was evaluated using standard metrics:

- **Precision:** Measures the accuracy of the positive predictions.
- **Recall:** Measures the ability to capture all relevant positive instances.
- **F1-Score:** The harmonic mean of Precision and Recall.
- **Accuracy:** The overall correctness of the predictions.

B. Evaluation of Techniques

Using the 'ant' project, we generated ASTs and CFGs to capture the syntactic and control flow structures of the code, respectively. These representations, along with pre-trained models like Codex and CodeBERT, were used to predict software defects.

C. Results

The performance of the AST, CFG, and pre-trained model techniques is summarized in Table I. The results demonstrate that while pre-trained models offer the highest performance, ASTs and CFGs provide competitive results with significantly lower computational costs.

TABLE I
PERFORMANCE OF AST, CFG, AND PRE-TRAINED MODELS

Technique	Precision	Recall	F1-Score	Accuracy
AST	0.80	0.78	0.79	0.80
CFG	0.83	0.81	0.82	0.83
Pre-trained Models	0.90	0.88	0.89	0.90

D. Discussion

1) Comparison with Pre-trained Models: Pre-trained models like Codex and CodeBERT provide higher accuracy and F1-scores compared to ASTs and CFGs. However, they require significant computational resources, which can be a limitation in practical applications. ASTs and CFGs offer a more balanced approach, providing competitive performance with lower computational costs, making them suitable for resource-constrained environments.

2) Implications for Software Engineering: The structured representations provided by ASTs and CFGs are not only useful for defect prediction but also for other software engineering tasks such as code review and debugging. These techniques allow for a deeper understanding of the code's structure and control flow, which is essential for maintaining and improving software quality.

V. LIMITATIONS AND FUTURE WORK

While our results are promising, further research is needed to explore the scalability of these techniques to larger codebases and other software projects. Additionally, integrating ASTs and CFGs with other advanced techniques, such as data dependence graphs (DDGs), could further enhance their effectiveness.

A promising direction for future work is the exploration of traversal paths in Abstract Syntax Trees (ASTs). Traversal paths, such as root-to-leaf and leaf-to-leaf paths, can provide unique insights into the structural and semantic properties of source code. By analyzing different traversal paths, we can generate more detailed and context-aware representations of the code.

A. Traversal Paths in ASTs

1) Root-to-Leaf Paths: Root-to-leaf paths traverse from the root node of the AST to each leaf node, capturing the hierarchical structure of the code. This method can help in understanding the nested structure and the flow of control within the code. It is particularly useful for identifying deeply nested constructs that may indicate complex logic or potential defects.

2) Leaf-to-Leaf Paths: Leaf-to-leaf paths, on the other hand, focus on the relationships between leaf nodes, representing the connections between different terminal elements of the code. This approach can highlight dependencies and interactions between various parts of the code, which are crucial for detecting code smells and ensuring code maintainability.

3) Exploring Other Traversal Paths: In addition to root-to-leaf and leaf-to-leaf paths, other traversal methods, such as in-order, pre-order, and post-order traversals, can be explored. These methods may offer different perspectives and additional insights into the code structure and logic. Investigating these traversal paths could lead to the discovery of new patterns and improve the accuracy of defect prediction models.

By combining these traversal path analyses with ASTs and CFGs, we aim to enhance the granularity and depth of code representations, enabling the development of more sophisticated models for defect prediction and code quality assessment.

VI. CONCLUSION

Our study demonstrates that ASTs and CFGs are effective and computationally efficient techniques for software defect

prediction. While pre-trained models offer higher accuracy, the practical advantages of ASTs and CFGs make them a valuable tool in the software engineer's arsenal. Future work will focus on optimizing these techniques and exploring their application to a broader range of software engineering tasks, including the analysis of traversal paths in ASTs.

VII. Acknowledgment

I would like to thank my mentors, Prof. Saurabh Tiwari from Dhirubhai Ambani Institute of Information and Communication Technology and Prof. Santosh Singh Rathore from Atal Bihari Vajpayee Indian Institute of Information Technology and Management, for their invaluable guidance and support throughout this research.

References

[1] A. Abdu, Z. Zhai, H. A. Abdo, and R. Algabri, "Software defect prediction based on deep representation learning of source code from contextual syntax and semantic graph," *Journal of Software Engineering and Applications*, vol. 15, no. 7, pp. 285–299, 2022.

[2] X. Liu, H. Chen, H. Zhang, and Y. Jiang, "Deep learning for source code analysis: A survey," *IEEE Access*, vol. 9, pp. 21 821–21 835, 2021.

[3] X. Wang, Z. Li, J. Sun, and Y. Liu, "Contextual embeddings for software defect prediction: A comparative study," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 182–196, 2023.