The Journal of Systems and Software 217 (2024) 112156

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

SGT: Aging-related bug prediction via semantic feature learning based on graph-transformer☆

Chen Zhang [a,b], Jianwen Xiang [a,b,*], Rui Hao [a], Wenhua Hu [a], Domenico Cotroneo [c], Roberto Natella [c], Roberto Pietrantuono [c]

[a] School of Computer Science and Artificial Intelligence, Ministry of Education Engineering Research Center for Transportation Information and Safety, Hubei Province Key Laboratory of Transportation Internet of Things Technology, Wuhan University of Technology, Wuhan, China
[b] Wuhan University of Technology Chongqing Research Institute, Chongqing, China
[c] University of Naples Federico II, Italy

ARTICLE INFO

Keywords:
Aging-related bug prediction
Graph-transformer
Deep semantic feature
Code property graph
Software quality

ABSTRACT

Software aging, characterized by an increasing failure rate or performance decline in long-running software systems, poses significant risks including financial losses and potential harm to human life. This is primarily attributed to the accumulation of runtime errors, commonly referred to as aging-related bugs (ARBs). ARBP aims to detect and address ARBs before software release, optimizing testing resource allocation. However, ARBP's effectiveness relies heavily on dataset quality. Prior research often relied on manually designed metrics that lack semantic features, resulting in low prediction accuracy. Some studies construct models to learn semantic features from source code, but typically focus on token-level features from abstract Syntax Trees, neglecting critical topological and functional connections. In this paper, we introduce the SGT model, an ARBP method based on Graph-Transformer. This model efficiently extracts semantic information and logical structures from source code, capturing data dependencies and program dependencies. We also propose subgraph sampling based on node degree to reduce structural complexity and apply random oversampling to address class imbalance. Experiments on three projects demonstrate notable improvements. For instance, SGT achieved F1 scores of 0.726 and 0.706 on Linux and MySQL, respectively. Compared to ALW, SGT shows a 12.3% and 6.8% improvement on Linux and MySQL.

1. Introduction

With the increasing size and complexity of software systems, writing high-quality software within a limited time and resources becomes increasingly challenging. In such complex software systems, software aging is triggered more frequently due to highly intricate logical computations or nonstandard coding practices. Software aging is a phenomenon that occurs in long-running systems, leading to performance degradation and eventual system crashes. The primary cause is the triggering and propagation of aging-related bugs (ARBs) that manifest over time in such systems (da Costa et al., 2021). ARBs are often associated with issues like memory leaks, storage problems, socket anomalies, unreleased files, socket exceptions, and so on (Huang and Kintala, 1993). These problems can result in significant economic losses or even endanger human lives in the real world (Palmer, 2022). This issue has been observed in various systems, including Linux systems (Cotroneo et al., 2010), Web serves (Kalantari et al., 2020), Android systems (Cotroneo et al., 2022), Java Virtual machines (Ghanavati et al., 2020), and deep learning frameworks (Liu et al., 2022). Therefore, during the software testing phase, it is crucial to eliminate ARBs to prevent the occurrence of aging phenomena.

Aging-related bug prediction (ARBP) is an efficient way to identify ARBs during testing (Zhang et al., 2023), which examines the source code files of complex systems to identify the presence of ARBs. Typically, these systems consist of thousands of modules and millions of lines of code. The complexity metrics of software modules are extracted from the source code as features for machine learning. These features are then utilized to build machine learning models that predict whether new code or modules contain ARBs. Previous research in feature extraction can be divided into two methods: manually design-based software

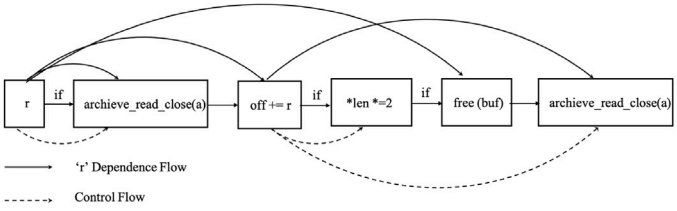
☆ Editor: Prof. Raffaela Mirandola.

* Corresponding author at: School of Computer Science and Artificial Intelligence, Ministry of Education Engineering Research Center for Transportation Information and Safety, Hubei Province Key Laboratory of Transportation Internet of Things Technology, Wuhan University of Technology, Wuhan, China.
E-mail addresses: zhangchenorange@whut.edu.cn (C. Zhang), jwxiang@whut.edu.cn (J. Xiang), ruihao@whut.edu.cn (R. Hao), whu10@whut.edu.cn (W. Hu), cotroneo@unina.it (D. Cotroneo), roberto.natella@unina.it (R. Natella), roberto.pietrantuono@unina.it (R. Pietrantuono).

https://doi.org/10.1016/j.jss.2024.112156
Received 16 February 2024; Received in revised form 26 May 2024; Accepted 8 July 2024
Available online 14 July 2024

```
. . .
for (;;) {
r = archive_read_data(a, buf + off, *len - off);
if (r == ARCHIVE_OK) {
archive_read_close(a);
*bufp = buf;
*len = off;
return 0;
}
. . .
off += r;
if (off == *len) {
*len *= 2;
if (*len < off) {
if (mflags.verbosity)
warnx("File too large: %s", file);
free(buf);
archive_read_close(a);
return -1;
}
buf = erealloc(buf, *len);
}
}
. . .
```

(a) Code slice from NetBSD



(b) Control and data flow for buffer *r* in (a)

AltAvgLineBlank	AltAvgLineCode	AltAvgLineComment	AltCountLineBlank	AltCountLineCode	AltCountLineComment	…	UniqueDerefUse	DerefSet	UniqueDerefSet
5	40	4	206	158	506	…	39	103	22

(c) Static code metrics for (a)

for_loop for condition init condition increment body assign var r call function archieve_read_data args var a expr var buf op + … call func erealloc args var buff var * len

(d) Semantic sequential tokens for (a)

Fig. 1. The motivation code slice of bug 51 040 from NetBSD dataset.

metrics extraction method (Qin et al., 2023) and machine learning-based deep semantic feature learning method (Zhou et al., 2022; Liu et al., 2020).

Manually design-based software metrics extraction methods extract features using tools like clang (Clang, 2016) or understood to assess source code quality. These metrics include measures of code size complexity, and software development processes, such as LOC, Halstead, McCabe complexity, CK metrics, code churn, micro-interactions, and so on. Some researchers (Qin et al., 2023) also integrate aging-related metrics (Cotroneo et al., 2013) with common metrics to construct ARBP models, or propose network features to model aging-related information flow in software. However, handcraft metrics often fail to distinguish codes with different semantics.

Machine learning-based deep semantic feature learning method is proposed to address these limitations by learning semantic and syntactic features from source code, enhancing the understanding of its logical structure. The most discussed feature extraction methods in software defect prediction (SDP) include token sequence-based (Zhou et al., 2022; Liu et al., 2020) and graph-based (Bryan and Moriano, 2023; Zhou et al., 2022; Tang et al., 2022) methods. The former extracts token sequences from code abstract syntax trees (Neamtiu et al., 2005), effectively bridging the semantic feature gap but lacking control flow and data relationships among code lines. Additionally, converting the AST into a sequential structure loses the structural relationships carried by the AST, weakening the understanding of the code's contextual relationships. The latter converts source code into graph representations (e.g., control flow graphs, data flow graphs), addressing the limitations of token sequence-based methods but may encounter complexity issues in large-scale projects. Although other methods treat source code and similar data as natural language, they do not discuss how to efficiently extract features in large-scale complex projects.

In Fig. 1(a), we present a code snippet that is misclassified by both traditional static code metrics-based methods and deep semantic token sequence-based methods. This code is sourced from Bug 51 040 in NetBSD. The issue is a memory leak in the *read_and_decompress* function, where allocated memory is not properly released after use or upon failure in subsequent operations. Specifically, the *read_and_decompress* function calls *archive_read_close* but fails to invoke *archive_read_finish*. The correct approach is to call *archive_read_finish* after the reading stream ends to release the resources allocated by libarchive, ensuring proper resource management and preventing memory leaks. The misclassification by existing methods can be attributed to the following reasons: traditional static code metrics-based methods are unable to capture the semantic context of the source code, as they only measure simple code features such as the number of functions, conditional statements, and lines of code (see Fig. 1(c)). Although token sequence-based deep semantic feature extraction methods overcome some limitations of traditional techniques (see Fig. 1(d)), they linearize the code structure into a sequential sequence of tokens. This approach overlooks control flow and other dependency features, leading to the incorrect assumption that an already opened file read object resource is closed without verifying if all associated resources have been released. Consequently, if the *read_and_decompress* function is called frequently, it poses a risk of memory overflow.

For this example, we hope to capture the control flow and data flow of the target program using graph features (see Fig. 3(b)). If we can achieve this, we can infer that the data dependency of *archive_read_close* is merely closing the resource without performing memory release operations (since it does not reflect the relevant edges in the data dependency graph), leading to memory overflow. However, most existing graph-based prediction methods are predominantly used in the field of vulnerability detection. Whether these graph feature extraction methods and prediction models can be directly applied to the ARBP task is unknown. This is because the ARBP task has more complex source code systems, which result in the generated graph becoming too large, leading to the model being unable to train effectively or even causing the training to crash.

To solve the above problems, this paper proposes an ARBP method via semantic feature learning based on the Structure-Aware Transformer (SAT) (Chen et al., 2022), which we named the SGT model. The SGT model firstly leverages code property graphs (Yamaguchi et al., 2014) to describe the source code, which can effectively extract the control flow and other dependency features to enhance the model's feature representation capability. Then, the sub-graph sampling method based on node degree is devised to tackle the complexity of graph structures in complex code systems, which can reduce the structural features of the original graph to retain only key nodes and branches with high utility. At last, random oversampling is employed to increase the number of ARB-prone instances, mitigating the risk of model ineffectiveness during model training. Finally, the extracted features are faded into the Graph Convolutional Network (GCN) (Liu and Zhou, 2022) with a Structure-Aware Transformer (SAT) (Chen et al., 2022) to capture contextual information in the source code effectively. Our contributions are summarized as follows:

- We propose a novel graph model, which we named the SGT model designed to extract features from the source code of complex software systems, utilizing code property graphs to capture both semantic information and logical structures proficiently. This is the first exploration of the applicability of graph models to code files in complex systems.

• We employ a sub-graph sampling method based on node degree, retaining the core nodes and branches with high relationships of the source code while reducing computational complexity. In addition, random oversampling is employed to increase the number of ARB-prone instances, mitigating the risk of model ineffectiveness during model training.

• Experimental results on three datasets demonstrate the applicability of the SGT model. In comparison to traditional static code metrics approaches and deep semantic feature learning methods, our model significantly improves the overall predictive performance, particularly by reducing the model's F1 score. For instance, compared to the SOTA deep semantic feature learning method ALW, SGT achieved a 12.3% improvement on Linux and a 6.8% improvement on MySQL.

The remainder of this paper is organized as follows. The preliminary of this work is introduced in Section 2, and the proposed SGT model is detailed in Section 3. Section 4 and Section 5 introduce the experiment setup and results. Section 6 introduces the threats to validity. The related works about ARBP using static code matrics and semantic feature learning are discussed in Section 7. Finally, we address the conclusion and future work in Section 8.

2. Preliminary

To facilitate the integration of source code into machine learning models for detecting ARBs, an intermediate representation is required to transform the code into vectors while preserving semantic information. Common methods for this purpose include token sequence representation (Wang et al., 2018), abstract syntax trees (Neamtiu et al., 2005; Allen, 1970), program dependency graph (Ferrante et al., 1987), code property graph (Yamaguchi et al., 2014), etc. This study adopts code property graphs as the intermediate semantic representation and focuses on elucidating the theoretical underpinnings of their components, which are derived from abstract syntax trees, control flow graphs, and program dependence graphs. Furthermore, graph neural networks (Liu and Zhou, 2022) serve as the semantic learning framework, and thus, an overview of their fundamental constructs is provided alongside the aforementioned feature representations.

2.1. Representations of source code

Abstract syntax trees (AST) (Neamtiu et al., 2005) typically serve as the initial intermediate representations generated by the compiler's code parser, forming the basis for generating many other code representations. AST trees represent programs by encoding the nesting of statements and expressions in the code. It effectively displays the complete static information of the source code, with a clear hierarchy that facilitates traversal and manipulation, making them widely used for identifying semantically similar code (Baxter et al., 1998). However, abstract syntax trees do not capture the specific syntax of a program. For instance, in the C language, a comma-separated list of declarations typically results in the same abstract syntax tree as two consecutive declarations. Furthermore, they are not suitable for more advanced code analyses, such as detecting dead code or uninitialized variables. The reason for this limitation is that control flow and data dependency relationships are not explicitly represented through this code representation.

Control Flow Graphs (CFG) (Allen, 1970) depicts the execution paths and control dependencies in a program, determining the sequence of statement executions and the order of procedure calls. CFG construction relies on AST trees, initially incorporating structured control statements such as if, $while$, for, $switch$, and subsequently refining the graph with non-structured control statements like $goto$, $break$, and $continue$. Conditional statements govern path selection, with nodes representing both statements and conditions, interconnected by directed edges denoting control flow. Unlike the AST, CFG edges are labeled true or false to indicate data flow, providing insight into the real-time execution process of a procedure by depicting all potential flows of basic block executions. The CFG serves as a comprehensive representation of a program's semantic logic, mitigating the impact of code obfuscation and finding applications in secure environments. It has been utilized in various contexts, including the detection of variants of known malicious applications (Gascon et al., 2013) and as a guiding tool for fuzz testing (Sparks et al., 2007). Additionally, CFG has become a standard in reverse engineering for enhancing program comprehension. However, while CFG effectively exposes the control flow of an application, it lacks precision in providing detailed data flow information.

Program Dependence Graph (PDG) is a graphical representation illustrating dependencies among program components, such as modules, classes, functions, and variables. It is structured as a labeled directed multi-graph, where nodes represent program components and edges signify dependencies. PDG construction relies on the program's CFG and integrates representations of both data and control dependencies. The data dependence graph delineates data constraints through edges reflecting data dependency paths, while the control dependence graph imposes constraints on statement execution by connecting paths with control dependence edges. PDG was first introduced by Ferrante et al. (1987) for dynamic program slicing to facilitate focused program analysis, and has found widespread application in software testing and analysis. For instance, Hammer and Snelting (Hammer and Snelting, 2009) utilized PDG for program information flow analysis, while Baah et al. (2008) devised a probabilistic PDG model for program fault localization and comprehension. However, in extensive programs featuring numerous modules, classes, and functions, PDGs may become highly intricate, posing challenges for comprehension and analysis.

Code Property Graph (CPG), proposed by Yamaguchi et al. (2014), integrates AST, CFG, and PDG into a unified data structure, capturing semantic details like control and data dependencies alongside code syntactic structures. It serves as a comprehensive abstract graph, adept at extracting semantic, syntactic, and structural code information. Initially, lexical and syntax analyses break down the code into tokens and organize them into a syntax tree. Semantic analysis follows, extracting additional details such as variable types and function call relationships, focusing on type inference, symbol resolution, and scope analysis. CPG is then constructed as a multi-level representation of entities and relationships, with attributes added to nodes and edges for further analysis of code characteristics and behavior. The CPG effectively addresses the challenge of adequately representing diverse vulnerability types within a single graph structure. For instance, the "lack of input validation" vulnerability is reflected in the CFG by the absence of branch conditions, a characteristic not adequately represented in the PDG. Similarly, the "division by zero" error is manifested in the AST by having the denominator of a division operation as zero, a feature not effectively represented in the CFG. By defining vulnerability patterns for different types and conducting graph traversal on the CPG based on these patterns, this approach enables the detection of various vulnerabilities. Due to its comprehensive integration of information from multiple program perspectives, the CPG demonstrates robust adaptability in identifying buffer overflows, integer overflows, memory disclosures, and format string vulnerabilities (Xu et al., 2022).

2.2. Deep graph models

Graph Neural Networks (GNN) are designed to leverage deep neural networks for the analysis of structured data (Scarselli et al., 2008; Xia et al., 2021). Despite the success of traditional deep learning methods in extracting features from Euclidean spatial data, challenges arise in scenarios where real-world data is generated from non-Euclidean spaces. Such data is often abstracted into graph structures, such as social networks, the World Wide Web, and similar contexts. Research indicates that the performance of conventional deep learning approaches in handling non-Euclidean spatial data, specifically graph structures, is

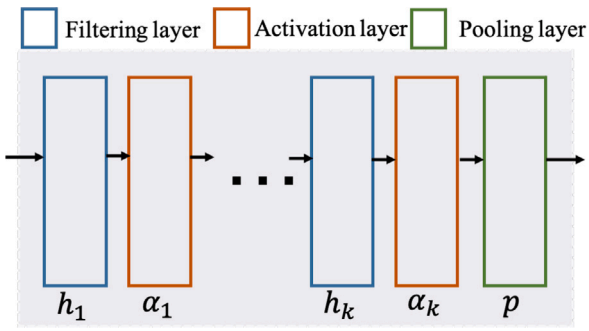


Fig. 2. A block in the graph neural network.

suboptimal (Zafeiriou et al., 2022). GNN (Liu and Zhou, 2022), as an influential method for deeply representing graph data, has exhibited outstanding capabilities in the analysis of graph-structured data. It can utilize deep neural networks to learn node representations from node features and graph structures.

Given the graph data G, its graph structure can be represented as $G = \{v, \varepsilon\}$, where $v = \{v_1, \ldots, v_N\}$ is a node set of size $N = |v|$, and $\varepsilon = \{e_1, \ldots, e_M\}$ is an edge set of size M. In graph classification tasks, the goal of graph neural network is to learn representative features (node features and graph structures) of the entire graph to classify graph data. Generally, a graph neural network consists of three parts: graph filtering layers, activation layers, and graph pooling layers. The graph filtering layers and activation layers are used to process features to obtain refined and meaningful features guiding the classification. The graph pooling layer summarizes node features, generating high-level features capable of capturing information from the entire graph. Typically, the graph pooling layer follows a series of graph filtering layers and activation layers. After the graph pooling layer, a coarsened graph is produced with more abstract and higher-level node features. The combination of these layers is referred to as a block, as shown in Fig. 2, where h_i, α_i and p represent the ith filtering layer, activation layer, and pooling layer in this block.

The input is the adjacency matrix $A^{(ib)}$ and feature matrix $F^{(ib)}$ of a graph $G_{ib} = \{v_{ib}, \varepsilon_{ib}\}$, and the output adjacency matrix $A^{(ob)}$ and feature matrix (include node features and edge features) $F^{(ob)}$ is the newly generated coarse graph of $G_{ob} = \{v_{ob}, \varepsilon_{ob}\}$. The computational process in this block is described as $F^{(i)} = h_i(A^{(ib)}, \alpha_{i-1}(F^{(i-1)})), i = 1, \ldots, k$, $A^{(ob)}, F^{(ob)} = p(A^{(ib)}, F^k)$. While, α_i represents the activation function of the ith layer, where α_0 is the identity function and $F^{(0)} = F^{(ib)}$. The entire Graph Neural Network framework can include one or more blocks. The computation process of a Graph Neural Network framework with L blocks can be represented as $A^{(j)}, F^{(j)} = B^{(j)}(A^{(j-1)}, F^{(j-1)}), j = 1, \ldots, L$, where $F^{(0)} = F$ and $A^{(0)} = A$ represent the initial node features and adjacency matrix of the original graph, respectively.

3. Methodology

This section introduces the SGT model, a semantic feature learning model based on an improved graph transformer, with three main components: graph-level feature extraction, data preprocessing, and SGT model construction. In the graph-level feature extraction section, we parse source code using Joern[1] for code property graph features and store it in Neo4J graph database (source code parsing). Subsequently, we construct the above features with DGL (Wang et al., 2019). Then, the Word2Vec model (Church, 2017) is applied to map nodes to low-dimensional space. In the data preprocessing section, we employ a sub-graph sampling method based on node degree to reduce the number of nodes in each graph. We define a predefined threshold α_d to determine which nodes are useless and need to be removed. Additionally, random oversampling is applied to address the severe class imbalance problem. In the SGT model construction, we combine an improved graph transformer block which is named Structure-Aware Transformer (SAT) with a graph neural network for model training, aiming to capture ARB codes with semantic features. Fig. 3 provides an overview of the SGT model.

3.1. Graph-level feature extraction

3.1.1. Source code parsing

The source code contains significant semantic information that surpasses the capabilities of a simplistic graph structure. While Abstract Syntax Trees (AST) adeptly capture syntax nuances, Control Flow Graphs (CFG) and Program Dependence Graphs (PDG) provide a mechanism for representing the source code's semantic complexities, embracing elements such as control flow, data dependencies, and control dependencies. The Code Property Graph functions as an all-encompassing graph structure, synthesizing various facets from AST, CFG, and PDG. This integration results in a comprehensive representation that spans both syntactic and semantic dimensions of the source code. We utilize the static analysis tool Joern for source code parsing, facilitating the generation of a code property graph. Joern is an open-source static analysis tool tailored for in-depth analysis of C/C++ source code. Leveraging this tool, we parse the source code into a code property graph, abstract syntax tree, control flow graph, and program dependence graph.

An example of a code property graph is shown in Fig. 4(b) for the code slice given in Fig. 4(a). For simplicity, property keys and values as well as labels on AST edges are not shown. In the example, the source code undergoes lexical analysis, transforming it into a sequence of tokens. Subsequently, through syntax and semantic analysis, these tokens are processed to generate an abstract syntax tree (depicted by the green branches in Fig. 4(b)), which accurately represents the syntactic structure of the code. This tree effectively conveys the logical execution order of the code, which is useful for determining the wrong order of operands. Building upon the abstract syntax tree, a control flow graph is then constructed by incorporating both structured (e.g., if, while, for) and unstructured (e.g., goto, break, continue) control statements, as shown by the red branches in Fig. 4(b). This graph elucidates the semantic logic of the code. Finally, a program dependency graph is derived from the control flow graph (illustrated by the blue branches in Fig. 4(b)), capturing the data and control dependencies within the program, which is important for capturing control and data flow patterns for ARBs like memory overflow in Fig. 1(a). Integrating these three representations, the final code property graph is meticulously organized and generated.

The generated code property graph will be stored in the Neo4J graph database. This storage approach enhances subsequent queries by exploiting the inherent information within the graph, facilitating the encoding of data into tensor formats. Neo4J is an open-source graph database based on graph theory, storing data in graph structures and utilizing the Cypher query language for data processing and retrieval. Neo4J's graph database is a flexible and efficient tool applicable to various domains requiring the handling of complex relationships and queries. Leveraging its graph structures and Cypher query language, Neo4J facilitates seamless execution of complex relationship analyses on extensive datasets.

3.1.2. Graph data construction

After generating the CPG, we construct the graph from the parsed source code. Retrieve data from the Neo4J graph database and utilize the Deep Graph Library (DGL) (Wang et al., 2019) open-source library for graph data construction. DGL is a Python library designed for graph deep learning, offering a rich set of tools and interfaces for building,

[1] https://github.com/joernio/joern.

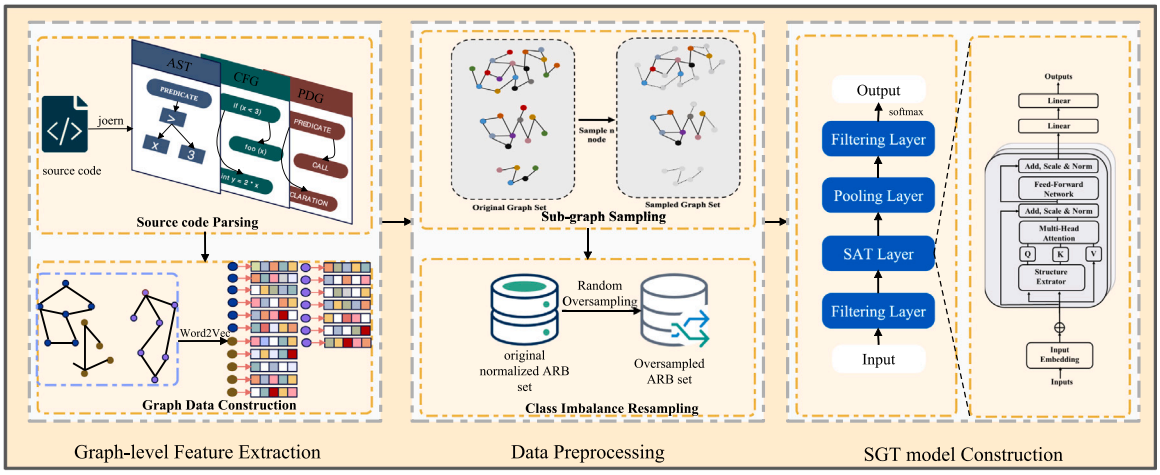


Fig. 3. The overall architecture of proposed SGT model.

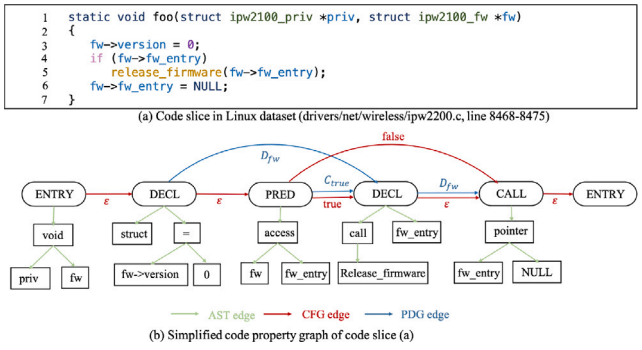


Fig. 4. Normalized code (a) and the parsed code property graph (b). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

training, and evaluating graph neural network models. Additionally, during the construction process, we assign different feature vectors to edges in the graph to accurately capture the interaction information between nodes.

Then, we perform graph embedding to map the nodes and edges into a low-dimensional vector space. For the node and edge features in the graph, the pre-trained Word2Vec model is utilized (Church, 2017) for vectorization. The Word2Vec model is designed to learn the semantic relationships between distinct words by analyzing their context in text. This model assigns a vector to each word based on the inferred relationships, ensuring that words with closer relationships in the text exhibit closer distances in their corresponding word vectors. Considering the lack of prior work discussing the appropriate vector length for the ARBP dataset, this paper conducts an in-depth experimental exploration of this aspect. This operation enables the representation of nodes and edges in a low-dimensional space, capturing semantic relationships between nodes and providing richer and more informative representations. The Pseudo-code for graph data construction is presented in Algorithm 1.

Algorithm 1: Graph Data Construction

input: .bin file
output: G: GDL graph data

1: **for** $edge$ in $bin.edges$ **do**
2: $src, dst = edge.src, edeg.dst$;
3: $src\_index = bin.nodes.index(src)$;
4: $dst\_index = bin.nodes.index(dst)$;
5: $edge\_type = preprocess(edge.attribute)$;
6: $G.edata['w'] = edge_type$;
7: **for** $node$ in $bin.nodes$ **do**
8: $node\_index = bin.nodes.index(node)$;
9: $node\_attr = node.attribute$;
10: $node\_feature = Word2vec(node.attr)$;
11: **return** G;

3.2. Data preprocessing

3.2.1. Sub-graph sampling

In typical software systems, the source code files we encounter often exceed a thousand lines of code, resulting in graph data with thousands or more nodes. The intricate structure of these graphs can significantly increase memory usage and computational complexity during feature learning, potentially affecting model performance. Moreover, we observe considerable variations in the lengths of different source code, leading to diverse graph sizes and posing challenges for feature learning. To address these issues, we employ a sub-graph sampling strategy inspired by Bhatia and Rani (2017) to reduce redundant nodes while retaining essential information in each graph. Specifically, we adopt a node-degree-based sampling method, selecting nodes based on their degrees in the large graph to preserve the network distribution and other topological structures. This approach helps maintain critical node attributes, facilitating improved identification of influential spreading nodes in the network (Salamanos et al., 2017). The detailed methodology is presented below:

Given the graph $G = \{V, \varepsilon\}$, the degree of a node v is defined as the number of edges associated with the node (Liu and Zhou, 2022):

$$degree(v_i) = \sum_{v_j \in V} f_\varepsilon(\{v_i, v_j\}) \quad (1)$$

where f_ε represents a function of edge weights or connection density. Next, each node is sorted in descending order based on its degree, resulting in a list of nodes. According to the predefined threshold α,

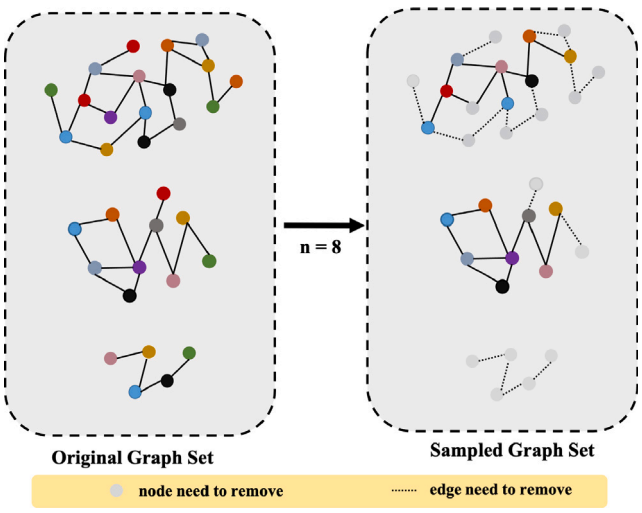


Fig. 5. The node-degree-based sub-graph sampling process. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the top n nodes with higher degrees are selected as the node set for the sub-graph. Where the α is a node degree that needs we defined, we think that if the node degree is greater than α, this node should be retained, otherwise it need not be removed.

And the selected nodes V' are denoted as:

$$V' = \{v_i \in V \mid degree(v_i) \geq \alpha\} \tag{2}$$

Here, V' represents the selected node set of the sub-graph. Finally, using the selected node set V', extract the corresponding edges from the original graph to construct the sub-graph $G' = \{V', \varepsilon'\}$:

$$G' = \{G_i \in G \mid |V_i| \geq \alpha'\} \tag{3}$$

The sub-graph G' contains edge information related to nodes with high degrees in the original graph. The core idea of this algorithm is to select important nodes in the graph by considering their degrees and then construct a sub-graph containing these crucial nodes to reduce the complexity of the graph. Such a sub-graph can alleviate computational burdens while preserving essential information. The performance and effectiveness of the algorithm may depend on the chosen degree threshold α and the number of nodes in the sub-graph, denoted as n. For ease of computation, we calculate the node degrees for all nodes, sort them in descending order, and select only the top n nodes as the sampled sub-graph, so we just need to talk about how n affects the model. For sub-graphs with fewer than n nodes, we consider them as ineffective graph representations and remove them. This is because the graph is too small, containing limited information, making it unable to learn an effective graph representation. On the other hand, if the graph is too large, with a complex structure, the computational complexity becomes too high, preventing effective parameter fitting.

The sub-graph sampling process is illustrated in Fig. 5. The left panel shows the original graphs designated for sampling, ranging from complex structures to simpler ones. For instance, the top graph represents intricate source code structures with thousands of nodes, depicted here in simplified form. The subsequent graphs depict progressively simpler structures. In the example provided, the original graphs contain 18, 10, and 5 nodes, separately. Given that the sampled sub-graph has $n = 8$ nodes, we sort all nodes by degree in descending order and select the top 8 nodes for each graph. The sampled nodes are depicted in the right panel, with colored nodes indicating the final selected nodes and dashed edges representing redundant edges. It is noteworthy that the value of n varies across different datasets and needs to be adjusted based on specific requirements. Through this sampling strategy, we selectively preserve influential nodes in the graph while discarding less impactful ones. This not only reduces computational burdens but also enhances scalability, ensuring effectiveness across diverse code sizes and improving overall model performance.

3.2.2. Class imbalance resampling

The class imbalance phenomenon, as defined by some studies (Ali et al., 2019; Pandey and Tripathi, 2021), pertains to situations where certain classes are significantly under-represented compared to others. This imbalance in the training dataset can adversely affect the performance of standard classifiers when applied to imbalanced scenarios, thereby impacting classification efficiency. In the context of ARB prediction, the class imbalance issue is particularly severe, which leads models to predict all files as ARB-free. Therefore, effective data augmentation methods are needed to enable the model to learn code features, particularly features related to ARBs. To address this, we employ the random oversampling (ROS) technique to balance the clean dataset for improved ARB prediction. ROS entails the replication of instances from the minority class, thus enhancing dataset balance and boosting model performance on the under-represented class. We chose ROS because it is widely used in many ARBP methods (Zhou et al., 2022; Liu et al., 2020; Zhang et al., 2023; Xu et al., 2020; Kumar and Sureka, 2018). We aim to overcome the classification bias introduced by minority class samples through a simple class imbalance sampling technique.

It is important to note that we first divided the dataset and applied ROS only to the training set. We adopt a stratified sampling approach to divide the data into training, validation, and testing sets. This method is commonly employed in software defect prediction to ensure that the ratio of minority and majority classes remains consistent across both sets (Zhang et al., 2023; Zhao et al., 2021). To maintain the class imbalance in the original data, we randomly select defective and non-defective samples from the original dataset according to the specified partition ratios.

3.3. SGT model construction

To better extract complex semantic information from the source code, this paper adopts the Graph Convolutional Network (GCN) (Liu and Zhou, 2022) as a backbone classification model to map the source code to a vector space. GCN can learn the relevant semantic information in the graph data. In addition, considering that ARBs typically occur after the system has been running for a long time, resulting from the accumulation of errors like memory leaks and storage problems. Consequently, the source code of ARB files may pass initial software testing, resulting in similarities in characteristic representations with ARB-free files. To effectively capture contextual information in the source code and learn the triggering conditions of ARBs, we apply an auto-encoder layer based on the improved graph transformer which is named Structure-Aware Transformer (SAT) (Chen et al., 2022) combined with GCN to construct our classification model, which we named SGT.

In our work, ARBP can be defined as a graph classification task. The classification of unlabeled graphs is classified by learning the labeled graphs. During the extraction of graph-level features from the source code, the features undergo iterative updates, incorporating the attributes of neighboring nodes to produce a refined representation of each node. This process integrates information related to both upstream and downstream nodes into the graph's feature representation. Given the training data $D = \{G_i, Y_i\}$, where Y_i represents the corresponding label of the graph, with $Y_i = 0$ indicating the sample G_i is ARB-free, and $Y_i = 1$ indicating a sample G_i with ARBs. The task objective is to train a model on the training set D to perform well in predicting unlabeled graphs. In our work, GNN is regarded as a feature encoder, mapping its input graph to a feature representation f_G:

$$f_G = GNN_{graph}(g, \theta_1) \tag{4}$$

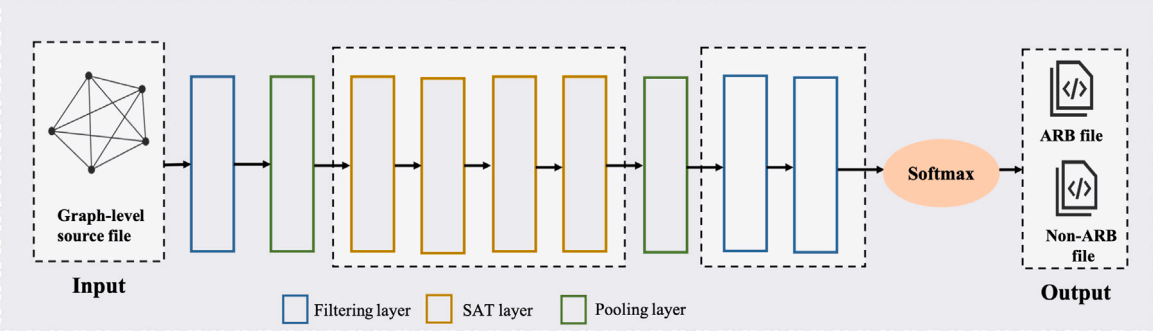


Fig. 6. SGT framework.

where GNN_{graph} is the graph neural network model learning graph-level representations, composed of SAT layer (Chen et al., 2022), pooling layer, and linear layer, as illustrated in Fig. 6. θ_1 represents the hyperparameters to be learned in the graph neural network. $f_G \in \mathbb{R}^{1\times d_{out}}$ is the generated graph-level representation, i.e., the desired deep semantic features. Subsequently, this graph-level representation is employed for graph classification: z_G:

$$z_G = softmax(f_G\theta_2) \tag{5}$$

where $\theta_2 \in \mathbb{R}^{d_{out}\times C}$ denotes the matrix for transforming graph representation into dimensions equal to the number of classes C, $z_G \in \mathbb{R}^C$ represents the predicted probability for input graph G. It is worth noting that this task is a binary classification problem, thus $C = 2$. In summary, the integration process of graph classification can be generalized as:

$$z_G = f_{GNN}(G, \theta) \tag{6}$$

where f_{GNN} encompasses all processes for f_G and z_G, and θ includes θ_1 and θ_2 which can be learned by minimizing the objective function:

$$L_{train} = \sum_{G_i\in D} l(f_{GNN}(G_i, \theta), Y_i) \tag{7}$$

where $l(\cdot,\cdot)$ denotes the cross-entropy loss. It is noted that, depending on different classification requirements, C can be set to other values.

In this task, SGT model employs average pooling operation $f_G = mean(F^{(ip)})$, where $mean()$ is applied to each channel $f_G[i'] = mean(F^{(ip)}_{:,i'})$, and $F^{(ip)}_{:,i'}$ represents the i'-th channel of $F^{(ip)}$. The linear layer of the model utilizes a fully connected layer to linearly transform node information, predicting whether the source code file contains ARBs. The SAT layer is an auto-encoder model based on the improved graph transformer (Chen et al., 2022). This model addresses the drawback of the traditional graph transformer, where the encoded node representations may not necessarily capture the similarity between nodes. The SAT model introduces the Structure-Aware Transformer, which is sensitive to the graph structure. This sensitivity is beneficial for capturing defect codes with similar declarations and facilitates the capture of semantically similar logic codes. The SAT model extracts sub-graph representations centered around each node before computing attention, incorporating structural information into multi-head self-attention. This enables position encoding for the source code. The SAT model aggregates local neighborhood information in the message-passing mechanism and captures structural interactions between nodes through a single self-attention layer. The definition of structural-aware attention is given by:

$$SA_{atten}(v) = \sum_{u\in V} \frac{K_{graph}(S_G(v), S_G(u))}{\sum_{w\in V} K_{graph}(S_G(v), S_G(w))} f(x_u) \tag{8}$$

Here, $S_G(v)$ represents the sub-graph centered around the node v associated with the node features X in the original graph. K_{graph} is any kernel comparing a pair of sub-graphs. This new self-attention function considers both attribute similarity and structural similarity between sub-graphs. Therefore, it generates node representations that are more expressive than the original self-attention. Additionally, this self-attention is no longer equivalent to any permutation of the nodes and is only equivalent to nodes coinciding with features and sub-graphs.

Table 1
Datasets description in our experiment.

Dataset	Language	Files	ARB file	Defective%
Linux	C	2191	20	0.91%
MySQL	C/C++	198	36	18.18%
NetBSD	C/C++	1534	19	1.24%

4. Experiment setup

This section introduces the datasets we used and the metrics for comparative experiments. Then, the evaluation metrics and baseline methods we discussed in our experiments are displayed.

4.1. Datasets

Dataset description: The study focuses on three datasets: Linux,[2] MySQL,[3] and NetBSD,[4] which are known for their large, complex, and long-lived software systems. Linux and MySQL are widely used open-source software systems with applications spanning various domains. For Linux, the study concentrates on four critical components: Linux-driver-net and Linux-driver-scsi, Linux-ext3, and Linux-ipv4. For MySQL, the study analyzes three critical components: InnoDB Storage Engine, Replication, and Optimizer part. NetBSD is collected by Zhang et al. (2023), which is a free and highly portable open-source operating system based on UNIX, which is known for its code quality and accuracy, as well as its widespread application. The study analyzes versions 7.0, 7.0.1, 7.0.2, 7.1, 7.1.1, 7.1.2, and 7.2 of NetBSD and selects five subsystems: kern, fs, usr.bin, usr.sbin, and crypto-dist, which are representative and practical for analysis.

It is noteworthy that, during the transformation of source code into graph data, certain files are excluded from the process due to their inability to generate meaningful graph representations. For instance, files containing pure virtual functions and function declarations, as depicted in Fig. 7(a), are removed. These files are often found in C++ class definitions within the Linux kernel, and typically lack intuitive structural information when represented graphically, particularly when serving as interface definitions. Similarly, files containing copyright statements and dealing with IPsec policy handling, illustrated in Fig. 7(b), are excluded. An example is the NetBSD kernel module, which involves the initialization of kernel modules and IPsec policy handling, whose complex information may not be effectively conveyed

[2] Linux: http://www-128.ibm.com/developerworks/libary/lembdev.html.
[3] MySQL: http://www.mysql.com/why-mysql/marketshare/.
[4] NetBSD: http://www.netbsd.org/.

```
/******************************************************************************
 *
 * Copyright(c) 2009 - 2011 Intel Corporation. All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of version 2 of the GNU General Public License as
 * published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110, USA
 *
 * The full GNU General Public License is included in this distribution in the
 * file called LICENSE.
 *
 * Contact Information:
 *  Intel Linux Wireless <ilw@linux.intel.com>
 * Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497
 *
 *****************************************************************************/

#include <linux/module.h>

/* sparse doesn't like tracepoint macros */
#ifndef __CHECKER__
#include "iwl-dev.h"

#define CREATE_TRACE_POINTS
#include "iwl-devtrace.h"

EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_iowrite8);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_ioread32);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_iowrite32);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_rx);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_tx);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_ucode_event);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_ucode_error);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_ucode_cont_event);
EXPORT_TRACEPOINT_SYMBOL(iwlwifi_legacy_dev_ucode_wrap_event);
#endif
```

(a) Linux

```
#ifdef HAVE_PFKEY_POLICY_PRIORITY
        if (xpl->sadb_x_policy_priority == 0)
        {
                priority_offset = 0;
                priority_str = "";
        }
        /* find which constant the priority is closest to */
        else if (xpl->sadb_x_policy_priority <
                 (u_int32_t) (PRIORITY_DEFAULT / 4) * 3)
        {
                priority_offset = xpl->sadb_x_policy_priority - PRIORITY_HIGH;
                priority_str = "prio high";
        }
        else if (xpl->sadb_x_policy_priority >=
                 (u_int32_t) (PRIORITY_DEFAULT / 4) * 3 &&
                 xpl->sadb_x_policy_priority <
                 (u_int32_t) (PRIORITY_DEFAULT / 4) * 5)
        {
                priority_offset = xpl->sadb_x_policy_priority - PRIORITY_DEFAULT
                priority_str = "prio def";
        }
        else
        {
                priority_offset = xpl->sadb_x_policy_priority - PRIORITY_LOW;
                priority_str = "prio low";
        }

        /* fix sign to match the way it is input */
        priority_offset *= -1;
        if (priority_offset < 0)
        {
                operator = '-';
                priority_offset *= -1;
        }
        else
        {
                operator = '+';
        }
#endif
```

(b) NetBSD

```
class Rpl_filter
{
public:
  Rpl_filter();
  ~Rpl_filter();
  Rpl_filter(Rpl_filter const&);
  Rpl_filter& operator=(Rpl_filter const&);

  /* Checks - returns true if ok to replicate/log */

  bool tables_ok(const char* db, TABLE_LIST* tables);
  bool db_ok(const char* db);
  bool db_ok_with_wild_table(const char *db);

  bool is_on();

  /* Setters - add filtering rules */

  int add_do_table(const char* table_spec);
  int add_ignore_table(const char* table_spec);

  int add_wild_do_table(const char* table_spec);
  int add_wild_ignore_table(const char* table_spec);

  void add_do_db(const char* db_spec);
  void add_ignore_db(const char* db_spec);

  void add_db_rewrite(const char* from_db, const char* to_db);

  /* Getters - to get information about current rules */

  void get_do_table(String* str);
  void get_ignore_table(String* str);

  void get_wild_do_table(String* str);
  void get_wild_ignore_table(String* str);

  const char* get_rewrite_db(const char* db, size_t *new_len);

  I_List<i_string>* get_do_db();
  I_List<i_string>* get_ignore_db();

private:
  bool table_rules_on;

  void init_table_rule_hash(HASH* h, bool* h_inited);
  void init_table_rule_array(DYNAMIC_ARRAY* a, bool* a_inited);

  int add_table_rule(HASH* h, const char* table_spec);
  int add_wild_table_rule(DYNAMIC_ARRAY* a, const char* table_spec);

  void free_string_array(DYNAMIC_ARRAY *a);

  void table_rule_ent_hash_to_str(String* s, HASH* h, bool inited);
  void table_rule_ent_dynamic_array_to_str(String* s, DYNAMIC_ARRAY* a,
                                           bool inited);
  TABLE_RULE_ENT* find_wild(DYNAMIC_ARRAY *a, const char* key, int len);

  /*
    Those 4 structures below are uninitialized memory unless the
    corresponding *_inited variables are "true".
  */
  HASH do_table;
  HASH ignore_table;
  DYNAMIC_ARRAY wild_do_table;
  DYNAMIC_ARRAY wild_ignore_table;

  bool do_table_inited;
  bool ignore_table_inited;
  bool wild_do_table_inited;
  bool wild_ignore_table_inited;

  I_List<i_string> do_db;
  I_List<i_string> ignore_db;

  I_List<i_string_pair> rewrite_db;
};
```

(c) MySQL

Fig. 7. Code slice example from three datasets that should be removed.

in graph structures. Additionally, files managing replication and binary log policy filters, without involving intricate logical calculations or pointer releases, as shown in Fig. 7(c), are also removed. These files, usually requiring additional context and syntactic structure for meaningful graph data generation, are deemed unsuitable as standalone C++ code may not offer sufficient information. The data removed in this experiment mainly consisted of files of the mentioned types and the dataset details we used are presented in Table 1.

Metrics description for experiment: To discuss the differences in performance between the SGT model and models using traditional metrics, curated two widely used feature sets: a 52-dimensional set (Cotroneo et al., 2013; Chouhan and Rathore, 2021; Xu et al., 2020; Li et al., 2021) and an 82-dimensional set (Zhang et al., 2023; Kumar et al., 2023) from prior research. The 52-dimensional feature set encompasses static code metrics (introduced in paper (Cotroneo et al., 2013)), such as program size metrics, McCabe complexity, Halstead metrics, and Aging-Related Metrics. These metrics are easily obtainable and offer insights into specific characteristics of software systems. Program size metrics, including total lines of code, comments, or declarations, provide a broad estimate of software complexity and have demonstrated utility in defect prediction (Gong et al., 2022; Feng et al., 2021). McCabe's cyclomatic complexity metrics and Halstead metrics, assessing code based on path and operand/operator counts, are incorporated, with a focus on their relevance to aging-related bugs due to the distinctive feature of error propagation complexity (Kumar and Sureka, 2018). Aging-related metrics, consisting of six program indicators related to memory management and data structure, are included to augment performance. The chosen aging-related metrics concentrate on primitives associated with memory allocation and filesystem access, as their presence may elevate the probability of resource leakage. For the 82-dimensional feature set, integration of the 52-dimensional set with an additional 30-dimensional feature set (introduced in paper (Zhang et al., 2023)) forms the basis of the prediction model.

4.2. Evaluation metrics

To evaluate the performance of the prediction model, which is a typical binary classification task, we employ five evaluation metrics, including Area Under the receiver operating characteristic Curve (AUC), $F1$, $Balance$, $Precision$, and MCC. The associated results can be derived from the confusion matrix, as illustrated in Table 2. The matrix's columns represent the predicted class, while the rows denote

Table 2
Confusion Matrix.

Actual class	Predicted class	
	ARB-prone	ARB-free
ARB-prone	TP	TN
ARB-free	FP	FN

the actual class. TN is the true negative instances, FP is the false positive instances, FN is the false negative instances, and TP is the true positive instances. Based on the above confusion matrix, the evaluation metrics are as follows:

AUC: The Area Under the Receiver Operating Characteristic (AUC) is a performance metric that assesses a classifier's overall performance by considering the balance between Recall and the False Positive Rate (PF). A high AUC indicates a desirable performance, characterized by high Recall and low PF.

$Precision$: It reflects the classifier's accuracy and is determined by the ratio of correctly predicted positive samples (TP) to the total predicted positive samples ($TP + FP$). Essentially, it quantifies the proportion of positive predictions that are accurately identified. The formula for $Precision$ is given in Eq. (9):

$$Precision = \frac{TP}{TP + FP} \tag{9}$$

$F1$: The $F1$ score is a measure of the balance between Precision and Recall, calculated as the harmonic mean of these two metrics, as shown in Eq. (10). A higher $F1$ score is desirable, signifying enhanced performance in both Precision and Recall.

$$F1 = \frac{2 * Precision * Recall}{Preciison + Recall} \tag{10}$$

$Balance$: The $Balance$ measure seeks to balance the PD and PF, providing a comprehensive assessment of the classifier's overall performance. It is computed as follows:

$$Balance = 1 - \frac{\sqrt{(0 - PF)^2 + (1 - PD)^2}}{\sqrt{2}} \tag{11}$$

MCC: Matthews Correlation Coefficient (MCC) is a comprehensive metric that considers TP, FP, TN, and FN instances. It provides a measure of the correlation between the predicted and actual results, effectively addressing class imbalance in ARBP classification. With values ranging from −1 to 1, a higher MCC signifies superior classifier performance. The calculation formula for MCC is provided in Eq. (12).

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{12}$$

4.3. Baseline methods

To validate the performance of the SGT model, we compare our model with seven commonly used traditional feature learning models and two state-of-the-art deep semantic feature learning models. In the selection of traditional feature learning models, we employ six classifiers commonly used in ARBP to discuss performance differences (Feng et al., 2021; Gong et al., 2022; Li et al., 2021; Xu et al., 2020; Chouhan and Rathore, 2021), we follow existing methods' classifier choices to facilitate discussion. We integrating these commonly discussed classification models facilitates researchers in comparing traditional feature-based machine learning models with our method. Additionally, to assess the performance of our model, we compare it with the state-of-the-art traditional feature learning model CM (Qin et al., 2023). Choosing the best-performing works can demonstrate the performance differences between our feature extraction model and the current optimal traditional feature extraction model to the readers. For the selection of deep semantic feature learning models, we compare our approach with ABLSTM-WSO (Zhou et al., 2022) and CNN (Liu et al., 2020), two of the latest new works in the past few years. The detailed method descriptions are presented below. Choosing these methods can provide us with a comparison of the performance differences among the most advanced deep semantic feature extraction models currently discussed in research.

4.3.1. Manually design-based software metrics extraction method

(1) **Decision Tree (DT)** (Myles et al., 2004): DT is a tree-like structure. It selects optimal features to partition the data, constructing a tree-shaped model used for classifying or predicting new data. At each node, the algorithm divides the dataset into different subsets based on the values of a specific feature, creating branches in the tree. This process selects the best feature for partitioning according to certain criteria, such as information gain or Gini coefficient. Leaf nodes represent the final classification or prediction results.

(2) **Random Forest (RF)** (Kotsiantis et al., 2007): RF is an ensemble learning method, which is based on the construction of decision trees. Random forest constructs multiple decision trees, introducing randomness in the training process of each tree. The results of multiple trees are then combined through voting or averaging to perform classification or regression.

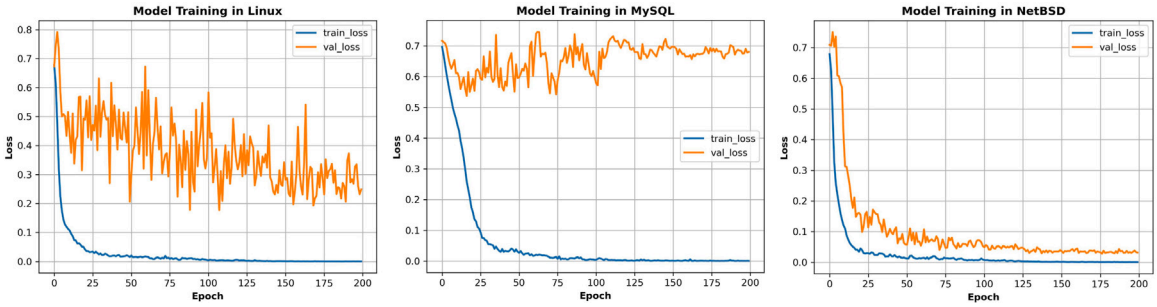
(3) **Naive Bayes Classifier (NB)** (Bhavsar and Panchal, 2012): NB is a probabilistic classification model utilizing Bayes' theorem, assuming strong independence between features. The classifier considers each file in the ARB dataset as independent and mutually exclusive. Naive Bayes can estimate parameter values, like mean and variance, with minimal training data.

(4) **Support Vector Machine (SVM)** (Bhavsar and Panchal, 2012): SVM is a supervised learning algorithm aiming to find a hyperplane in high-dimensional space for effective classification or regression. It maximizes the margin between instances of different classes during training, with support vectors being critical. SVM handles non-linear problems through kernel functions, mapping them to higher dimensions. The object of SVM is to maximize the margin for improved generalization to new data.

(5) **K-Nearest Neighbors Classifier (KNN)** (Bhatia et al., 2010): The idea of KNN is to identify the k nearest neighboring instances in the feature space for a given code file. The majority class among these neighbors determines the category of the code file, employing a voting mechanism.

(6) **Logistic Regression Classifier (LR)** (Kotsiantis et al., 2007): LR is a derivative of the linear regression model, utilizing the sigmoid function to map the linear range into the (0, 1) interval. It assigns weight factors (coefficients) to each feature of source code files, determining labels based on the product of feature values and corresponding weights. Coefficient optimization is the core task of this classifier.

(7) **CM** (Qin et al., 2023): was introduced by Qin et al. who proposed network features based on dependency graphs to model the propagation of aging-related information within the software. They devised specific metrics for ARBs by considering interactions related to aging between files. Initially, they constructed an aging-related dependency network, treating files as nodes and aging-related call dependencies between files as edges. Subsequently, network measures such as degree and closeness were extracted based on this network. Considering aging-related interactions during network construction allowed the extracted network measures to capture the flow of aging-related information and the overall topology of the aging-related network, aspects not effectively captured by traditional code metrics. To simplify the model, they utilized ReliefF (Robnik-Šikonja and Kononenko, 2003) to rank individual metrics based on their importance in distinguishing instances of different types in the training set. Ultimately, prediction models were built by integrating aging-related network measures with baseline 52-dimension metrics for the ARBP task.



(a) Loss on Linux (b) Loss on MySQL (c) Loss on NetBSD

Fig. 8. The loss for SGT model training on Linux, MySQL, NetBSD dataset.

4.3.2. Machine learning-based deep semantic feature learning method

(1) CNN (Liu et al., 2020): Liu et al. proposed a token sequence-based feature extraction method. They utilize the Clang tool (Clang, 2016) to extract AST features from the source code, and then token sequences are parsed and combined with six handcraft ARB metrics (Cotroneo et al., 2013) as the final features. To overcome the class imbalance problem, they adopt the random oversampling method to re-balance the training set. Finally, features are converted as vectors and fed into a convolution neural network for the ARBP task.

(2) ABLSTM-WSO (Zhou et al., 2022): abbreviated as ALW, was introduced by Zhou et al. In alignment with the approach proposed by Liu et al. (2020), ALW is a feature extraction method based on token sequences. However, Zhou et al. innovate by designing a deep neural network that integrates the bidirectional long short-term memory (BLSTM) and attention mechanism, aiming to extract context-sensitive semantic features from the code. Additionally, they employ a weakly supervised oversampling (WSO) method to address class imbalance challenges within datasets.

5. Experiment results and analysis

In this section, we report and analyze the experiment results. Our experiments are conducted on the Python3.7 platform, using a Ubuntu operating system with 6 GB memory, and an Inter Xeon Gold 5281R CPU @ 2.3 GHz processor with RTX4090-24G GPU. It should be noted that in all of the experiments, no manual parameter adjustment is carried out in the performance evaluation stage. All the experiment results try to answer the following five questions (RQs):

5.1. RQ1: How does SGT model performance?

Motivations and Approach. We propose the SGT model which is a semantic feature learning model based on the graph transformer. To validate whether our model can predict more ARBs, we visualize the training process of the model on three datasets and observe their violin plots to analyze the predictive distribution of model performance. To provide a more accurate evaluation of the overall model performance, we also report the average performance metrics on the three datasets. It is noteworthy that we observe the model's training to maintain almost ineffective predictive accuracy when not applying random oversampling. Therefore, no results are reported for this approach.

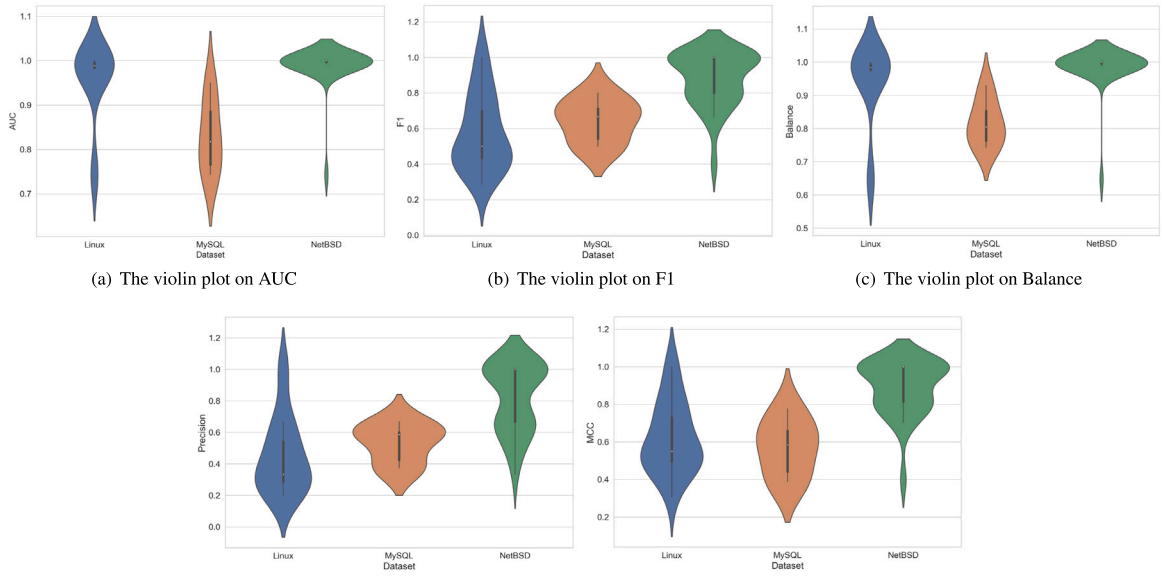
All experiments are independently repeated 30 times to avoid randomness, with each model trained for 200 epochs. The model is selected for the test when the loss is at its minimum. The data split ratio for training, testing, and validation sets is 8:1:1, and all experiments report the best results. Parameter choices for the Linux dataset are: 100 sub-graph sampling nodes and the node feature dimension is 256. For the MySQL dataset, 100 sub-graph sampling nodes are adopted and the node feature dimension is 16. On the NetBSD dataset, 100 sub-graph sampling nodes are chosen, and the node feature dimension is 128.

Results. All the experiment results are shown in Fig. 8, Fig. 9, and Table 3. Fig. 8 is the training process of the SGT model on three datasets, Fig. 9 is the violin plot, and Table 3 is the average performance of SGT, we report the AUC, F1, Balance, Precision, and MCC to evaluate the model performance in different views.

From Fig. 8, it can be observed that the training loss consistently decreases across all three datasets, indicating effective feature learning. Validation loss for the NetBSD dataset steadily declines, suggesting improved predictive accuracy for unseen data. However, fluctuations in validation loss for the Linux dataset are observed due to its high imbalance ratio, potentially limiting ARB-related information capture and compromising predictive accuracy. Despite this, the model remains effective, with persistent loss reduction. Conversely, the validation loss for MySQL decreases slower, This phenomenon is attributed to the comparatively smaller size of the MySQL dataset, causing underfitting and lower predictive accuracy compared to the other datasets.

From Fig. 9, the observation from the violin plots aligns with the above analysis, affirming the relatively focused predictive performance of the SGT model in the Linux and NetBSD datasets. The violin plots illustrate that various performance metrics exhibit more concentrated predictive outcomes in these datasets compared to the MySQL dataset. This indicates that the predictive capability of the SGT model is relatively stable on the Linux and NetBSD datasets. To accurately evaluate the model's performance across the three datasets, we present its average performance in Table 3.

From Table 3, we can conclude that the SGT model exhibits the optimal AUC (0.996) and Balance (0.994) on the Linux dataset, but it has the poorest F1 (0.726) and Precision (0.579), indicating lower predictive accuracy. We attribute this to the highest imbalance ratio in the Linux dataset, causing the model to predict all files as non-aging-related bugs. On the NetBSD dataset, the model achieves the best F1 (0.827), Precision (0.744), and MCC (0.838). However, on the MySQL dataset, none of the metrics reach the optimum values, indicating that the model can better learn data features in datasets with a larger number of files and lower imbalance ratios, resulting in stronger predictive capabilities. However, a comprehensive analysis of the metrics from three datasets indicates that the proposed model still possesses good defect semantic learning capabilities. This is because the graph features based on code property graphs can effectively capture the logical structural relationships in source code (related to the extraction of AST features). Furthermore, the control dependencies and program dependencies information extracted during the source code analysis phase also guide the SGT model for further semantic understanding. The capture of this information is beneficial for the SGT model to understand the mechanisms of ARBs from the context view, thereby enhancing the model's predictive capability.



(a) The violin plot on AUC (b) The violin plot on F1 (c) The violin plot on Balance

(d) The violin plot on Precision (e) The violin plot on MCC

Fig. 9. The violin plot on AUC, F1, Balance, Precision, and MCC for SGT model on Linux, MySQL, and NetBSD dataset. The white dot on each violin plot represents the medium.

Table 3
The average performance of the SGT model on the Linux, MySQL, and NetBSD datasets.

Datasets	AUC	F1	Balance	Precision	MCC
Linux	**0.996**	0.726	**0.994**	0.579	0.754
MySQL	0.919	0.796	0.878	0.714	0.704
NetBSD	0.980	**0.827**	0.973	**0.744**	**0.838**

Answer to RQ1: The SGT model performs well across all three datasets, particularly excelling on the Linux dataset with an AUC of 0.996 and a Balance of 0.994. On the NetBSD dataset, it achieves an F1 score of 0.827, a Precision of 0.744, and an MCC of 0.838. However, its performance may slightly degrade on highly imbalanced and small datasets. Thus, it is advisable to use the model on datasets with an adequate number of samples, considering its suitability for datasets with significant class imbalance.

5.2. RQ2: How does SGT performance compared with the state-of-the-art deep semantic models?

Motivations and Approach. In the field of ARBP, there have been several notable investigations focusing on deep semantic feature models, which have demonstrated commendable predictive efficacy. To evaluate the performance gap between SGT and state-of-the-art models in ARBP, we selected some recently prominent models in the field, namely, CNN (Liu et al., 2020) and ALW (Zhou et al., 2022). Since all models reported Balanced results, this section focuses on comparing this metric. It is worth noting that we observe a suboptimal predictive performance of the ALW method when utilizing random oversampling. To highlight this, we employed the weakly supervised oversampling method proposed in this paper and compared its optimal results with our model (SGT + random oversampling). Additionally, since the ALW method reports F1 results, we further assessed the differences between this model and SGT in terms of the F1 metric. The experiment details align with RQ1.

Results. All the experiment results are shown in Tables 4 and 5. Table 4 is the average performance on the Balance of various deep models on the Linux and MySQL datasets, and Table 5 is the average performance on F1 of various deep models on the Linux and MySQL datasets. In each method, we specified the models for comparison along with the classifiers they employed, which is 'method + classifier'.

Table 4
The average performance on Balance of various deep models on the Linux and MySQL datasets.

Methods	Linux	MySQL
ALW + MLP	0.896	0.833
ALW + KNN	0.871	0.822
ALW + LR	0.896	0.774
ALW + DT	0.740	0.782
ALW + NB	0.847	0.761
CNN + LR	0.880	0.762
SGT	**0.994**	**0.878**

From Table 4, we can observe that, for the Linux dataset, SGT achieves a Balance of 0.994, outperforming the best-performing comparison model ALW+MLP/LR with a Balance of 0.896. This is because the SGT method employs a deep semantic learning approach based on graph features combined with a random oversampling technique. In contrast, the ALW+MLP/LR model uses a learning method based on AST-based token sequences and employs a more advanced oversampling method than ours. This indicates that the SGT model can better capture the contextual information of the code during the feature learning stage, deeply understanding the mechanisms triggered by ARBs, and thus more effectively detecting ARBs, thereby more effectively detecting ARBs. This also indicates that extracting graph features from source code from different perspectives can better preserve the logical semantics and other information of the code, and can compensate for the deficiencies in feature expression related to code structure and program dependencies in token sequences. Similarly, on the MySQL dataset, although the SGT model's Balance (0.878) experiences a marginal decrease, it maintains its status as the leading model when compared to the ALW (the best is 0.833 using the MLP classifier) and CNN (0.762) methods.

From Table 5, we can observe that, for the Linux dataset, SGT achieves an F1 of 0.726, outperforming the best-performing comparison model ALW+DT with an F1 of 0.603, which improved by 12.3%. This indicates that SGT exhibits better performance in handling classification tasks, as the F1 score is a comprehensive measure of precision

Table 5
The average performance on F1 of various deep models on the Linux and MySQL datasets.

Methods	Linux	MySQL
ALW + MLP	0.551	0.728
ALW + KNN	0.537	0.642
ALW + LR	0.479	0.636
ALW + DT	0.603	0.700
ALW + NB	0.386	0.590
SGT	**0.726**	**0.796**

and recall. A higher F1 score suggests that the model performs well in maintaining a balance between Precision and Recall. Therefore, it can be inferred that SGT is more robust compared to ALW and CNN, capturing the relevant features of ARBs more comprehensively in the ARBP task. Similarly, on the MySQL dataset, the SGT model achieves an F1 score of 0.796, exhibiting a 6.8% improvement compared to the optimal ALW method (ALW+MLP).

Answer to RQ2: Compared to the current state-of-the-art ALW model, SGT demonstrates a 9.8% improvement in Balance (which is 0.994) and a 12.3% improvement in F1 (which is 0.726) on the Linux dataset. For the MySQL dataset, SGT demonstrates a 4.5% improvement in Balance (which is 0.878) and a 6.8% improvement in F1 (which is 0.796).

5.3. RQ3: How does SGT performance compared with the state-of-the-art traditional models?

Motivations and Approach. Most ARBP studies still use traditional manually designed metric features (Kaur and Kaur, 2022a,c; Li et al., 2021; Khanna et al., 2021; Qin et al., 2023). To compare our model with metric-based machine learning models, we evaluated SGT against six common classifiers: decision tree, random forest, naive Bayes, support vector machine, k-nearest neighbors, and logistic regression. We employed two widely discussed feature sets: a 52-dimensional set and an 82-dimensional set. Random oversampling was applied to address the class imbalance. Additionally, we compared our model with CM (Qin et al., 2023) to analyze differences in feature sets. Experimental implementation details align with RQ1.

Results. All the experiment results are shown in Tables 6 and 7, and 8. Table 6 is the average performance of all models that need to be compared on the Linux datasets, Table 7 is the average performance of all models on the MySQL datasets, and Table 8 is the average performance of all models on the NetBSD datasets. In each method, we specified the classifier for comparison along with the feature set they employed, which is 'classifier + feature set'. When the feature set is 52 means the classifier uses the 52-dimensional feature set, and 82 means it uses the 82-dimensional feature set.

In Table 6, it is evident that on the Linux dataset, SGT achieves an AUC of 0.996, exhibiting a significant 13.8% improvement over the best-performing comparison model LR (0.858 using the 52-dimensional feature set). The higher AUC for SGT suggests that it has a superior ability to discriminate between positive and negative instances compared to LR. SGT is more effective in ranking instances and assigning higher probabilities to positive instances than LR.

In terms of F1, SGT (0.726) demonstrates a notable 67.1% improvement compared to the best traditional model LR (0.055 using the 52-dimensional feature set), which signifies an enhanced capability in addressing the ARBP task. Conversely, LR with low F1 may indicate an imbalance between precision and recall in practical applications. In terms of the Balance, SGT (0.994) outperforms the best traditional model LR (0.822 using the 52-dimensional feature set) by 17.2%. This improvement is also 21.3% higher than the latest model CM (0.781). It indicates that SGT highlights its exceptional performance in addressing imbalanced class data. In contrast, LR and CM with low Balance suggest a weaker ability to predict a file whether has the ARB or not.

Table 6
The average performance of various traditional models on the Linux dataset.

Methods	AUC	F1	Balance	Precision	MCC
DT-52/82	0.812/0.795	0.042/0.042	0.766/0.754	0.022/0.021	0.113/0.108
RF-52/82	0.792/0.772	0.035/0.029	0.745/0.709	0.018/0.015	0.099/0.088
NB-52/82	0.704/0.729	0.029/0.031	0.639/0.681	0.015/0.016	0.071/0.079
SVM-52/82	0.838/0.850	0.044/0.042	0.796/0.801	0.022/0.021	0.121/0.122
KNN-52/82	0.539/0.525	0.040/0.030	0.362/0.340	0.025/0.020	0.039/0.026
LR-52/82	0.858/0.836	0.055/0.054	0.822/0.801	0.028/0.028	0.140/0.133
CM-52/–	–	–	0.781	–	–
SGT	**0.996**	**0.726**	**0.994**	**0.579**	**0.754**

Table 7
The average performance of various traditional models on the MySQL dataset.

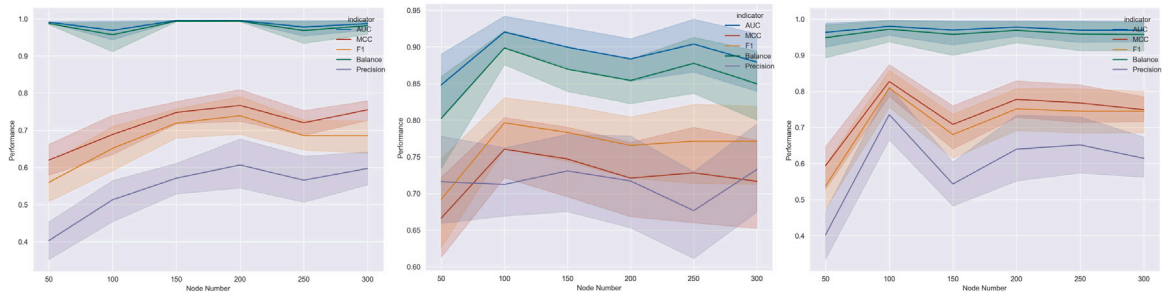
Methods	AUC	F1	Balance	Precision	MCC
DT-52/82	0.737/0.750	0.319/0.330	0.697/0.711	0.208/0.214	0.293/0.308
RF-52/82	0.740/0.771	0.301/0.332	0.710/0.745	0.186/0.208	0.279/0.319
NB-52/82	0.779/0.780	0.330/0.340	0.745/0.752	0.204/0.212	0.324/0.330
SVM-52/82	0.777/0.785	0.336/0.340	0.749/0.754	0.210/0.211	0.326/0.333
KNN-52/82	0.726/0.723	0.355/0.357	0.699/0.696	0.254/0.255	0.309/0.308
LR-52/82	0.768/0.750	0.338/0.330	0.749/0.731	0.214/0.212	0.321/0.304
CM-52/–	–	–	0.749	–	–
SGT	**0.919**	**0.796**	**0.878**	**0.714**	**0.704**

Table 8
The average performance of various traditional models on the NetBSD dataset.

Methods	AUC	F1	Balance	Precision	MCC
DT-52/82	0.609/0.602	0.041/0.035	0.555/0.550	0.025/0.019	0.055/0.047
RF-52/82	0.700/0.652	0.051/0.045	0.653/0.581	0.026/0.025	0.094/0.074
NB-52/82	0.675/0.667	0.040/0.039	0.628/0.621	0.021/0.020	0.076/0.072
SVM-52/82	0.701/0.708	0.048/0.047	0.667/0.668	0.025/0.024	0.091/0.092
KNN-52/82	0.542/0.527	0.053/0.043	0.380/0.358	0.035/0.029	0.045/0.031
LR-52/82	0.632/0.617	0.040/0.036	0.590/0.562	0.011/0.018	0.061/0.052
CM-52	–	–	0.781	–	–
SGT	**0.980**	**0.827**	**0.973**	**0.744**	**0.838**

For the Precision, SGT (0.579) exceeds the best traditional model LR (0.028 using the 52/82-dimensional feature set) by 55.1%. The higher Precision for the SGT model implies relative accuracy in predicting ARBs. In comparison, LR's Precision is extremely low, indicating more errors in its ARB predictions. Finally, in terms of the MCC, SGT (0.754) demonstrates a 61.4% improvement over the best traditional model LR (0.140 using the 52-dimensional feature set). The heightened MCC for SGT indicates its relative accuracy in predicting instances with and without ARBs. Conversely, LR's lower MCC may suggest more errors in its prediction.

Similarly, on the MySQL and NetBSD datasets, as evident from the results in Tables 7 and 8, SGT consistently achieves superior performance compared to all other models. From the overall experimental results, the deep semantic features used by the SGT model are more suitable for the ARBP task than the traditional features based on static code metrics. This is because traditional static code metrics typically focus on the surface attributes of code, such as the number of lines of code, number of functions, and number of variables. Although these metrics can quickly assess code complexity and potential problem areas, they do not delve into the semantic layer of the code, which is insufficient to understand the actual functionality and behavior of the code. Furthermore, static code metrics often lack an understanding of the code context, such as the interactions and dependencies between different parts of the code. They primarily provide isolated metrics



(a) Performance trend on Linux (b) Performance trend on MySQL (c) Performance trend on NetBSD

Fig. 10. The performance trend (in terms of AUC, F1, Balance, Precision, and MCC) on different graph node numbers.

without considering the overall software architecture or relationships between modules. Compared to deep semantic features, static metrics do not involve identifying advanced patterns, design patterns, or more complex logical structures in the code. Deep learning methods can capture these complex patterns and structures by learning from large samples of code. Additionally, static code metrics may not effectively identify new or subtle issues in the code, such as resource leaks that gradually emerge over time, which typically require deeper analysis to detect.

Answer to RQ3: Compared to the discussed traditional metric feature-based classification models and the state-of-the-art model CM, SGT outperforms all of the methods on three datasets. This indicates that semantic features based on code property graphs are more effective in representing source code.

5.4. RQ4: How does the sub-graph sampling impact the performance of the SGT model?

Motivations and Approach. In the SGT model, during the process of constructing graph data, we systematically reduce the number of nodes in each graph to a specified value denoted as n. However, the existing literature lacks exploration into how the number of nodes affects the model's performance, i.e., determining an optimal number of nodes for the model. To some extent, having a greater number of nodes implies a richer source of information, which could positively guide the model. Nevertheless, an excessive number of nodes may lead to intricate graph structures, resulting in heightened computational complexity and the introduction of redundant information, ultimately diminishing the model's performance. This experiment delves into the impact of varying node numbers on the model's performance. With all feature node dimensions set to 16, we select six sampling numbers ranging from low to high (50, 100, 150, 200, 250, 300) to observe the performance variations of the SGT model across three datasets. We report the AUC, F1, Balance, Precision, and MCC results. It is worth noting that we do not report the results conducted before imbalance sampling because we found that training the model directly with the original graph is ineffective. The model cannot be effectively trained due to the high-class imbalance, which makes it impossible to perform calculations and feature learning. The implementation details of the experiments align with RQ1.

Results. All the experiment results are shown in Fig. 10, Tables 9, 10, and 11. Fig. 10 is the model performance trend on different node numbers, we show the model performance using five evaluation metrics. It is worth noting that the displayed graph is an error band plot, where the solid the line represents the average value of 30 repeated experiments, and the shaded band above and below the line represents the value range of the corresponding method. Tables 9, 10, and 11 are

Table 9
The average predictive performance of the SGT model under different sub-graph sampling node counts in the Linux dataset.

Methods	AUC	F1	Balance	Precision	MCC
SGT-50	0.991	0.560	0.987	0.403	0.620
SGT-100	0.969	0.651	0.957	0.513	0.689
SGT-150	0.995	0.719	0.994	0.571	0.748
SGT-200	**0.996**	**0.764**	**0.994**	**0.641**	**0.788**
SGT-250	0.978	0.685	0.969	0.566	0.720
SGT-300	0.987	0.728	0.982	0.597	0.755

the average predictive performance of the SGT model under different sub-graph sampling node numbers in the Linux, MySQL, and NetBSD datasets, separately. In each method, we specified the SGT for comparison along with the sampling number we employed, which is 'SGT + subset sampling number'.

From Fig. 10, we can observe that, on the Linux dataset, the performance of the SGT model initially improves and then declines as the number of sampled nodes increases. The model achieves optimal performance when the number of nodes is 200. On the MySQL dataset, the AUC, F1, Balance, and MCC of the SGT model exhibit an initial improvement followed by a decline as the number of sampled nodes increases, with optimal performance achieved when the number of nodes is 100. However, the MCC reaches optimal performance when the number of nodes is 150. The Precision fluctuates with the increase in sampled nodes, achieving the optimal solution at n = 300. On the NetBSD dataset, the performance of the SGT model resemble those on the Linux dataset, with optimal performance achieved when the number of nodes is 100. To obtain more accurate model performance, we report the average results of the SGT model across different sampled node numbers.

From Table 9, we can observe that, on the Linux dataset, the SGT model attains its peak performance with a sub-graph sampling size of 200 nodes. It achieves the highest AUC of 0.996, an F1 score of 0.764, a Balance of 0.994, a Precision of 0.641, and an MCC of 0.788. However, deviations from this optimal sampling size, specifically with 50 or 300 sampled nodes, result in performance deterioration. This phenomenon can be attributed to the oversimplification of graph structures when sampled nodes are too few, leading to an inadequate representation of information. Conversely, an excessive number of sampled nodes introduces complexity to the graph structures, elevating computational demands and introducing redundant information, thereby diminishing the model's performance.

From Table 10, we can observe that, on the MySQL dataset, the SGT model achieves its optimal AUC value of 0.919, an F1 score of 0.796, and a Balance of 0.878 with a sub-graph sampling size of 100 nodes. However, when the sub-graph sampling size is 300 nodes, the model achieves its optimal Precision, which is 0.733. Additionally, with 150 sampled nodes, the MCC reaches its optimal value of 0.747. From Table 11, it is evident that, on the NetBSD dataset, the SGT model achieves its optimal AUC of 0.980, an F1 score of 0.809, a

Table 10
The average predictive performance of the SGT model under different sub-graph sampling node counts in the MySQL dataset.

Methods	AUC	F1	Balance	Precision	MCC
SGT-50	0.848	0.692	0.802	0.716	0.667
SGT-100	**0.919**	**0.796**	**0.878**	0.714	0.704
SGT-150	0.899	0.783	0.870	0.731	**0.747**
SGT-200	0.884	0.766	0.854	0.717	0.721
SGT-250	0.904	0.771	**0.878**	0.677	0.728
SGT-300	0.879	0.760	0.850	**0.733**	0.717

Table 11
The average predictive performance of the SGT model under different sub-graph sampling node counts in the NetBSD dataset.

Methods	AUC	F1	Balance	Precision	MCC
SGT-50	0.963	0.538	0.949	0.403	0.594
SGT-100	**0.980**	**0.809**	**0.972**	**0.736**	**0.827**
SGT-150	0.969	0.680	0.958	0.543	0.708
SGT-200	0.978	0.751	0.969	0.640	0.778
SGT-250	0.969	0.745	0.959	0.652	0.768
SGT-300	0.969	0.722	0.957	0.614	0.749

Balance of 0.972, a Precision of 0.736, and an MCC of 0.827 with a sub-graph sampling size of 100 nodes. The lack of clear regularity in the model's performance on the MySQL dataset can be attributed to the relatively fewer samples compared to the Linux and NetBSD datasets. This limitation may prevent the model's parameter learning from reaching an optimal solution, resulting in less stable performance. Generally speaking, considering the performance of the model across various datasets, a smaller number of nodes is sufficient to capture enough information to achieve good AUC and Balance while maintaining efficiency. However, to improve other metrics such as Precision and MCC, a larger number of nodes may be necessary.

Answer to RQ4: On the Linux and NetBSD datasets, the best sampling node numbers are 200 and 100. On the MySQL dataset, when the sampling node number is 100, the SGT model achieves optimal AUC, F1, and Balance. When the sampling node number is 300, the SGT model achieves optimal Precision. When the sampling node number is 150, the SGT model achieves optimal MCC.

5.5. RQ5: How does the feature dimensions of graph nodes impact the performance of the SGT model?

Motivations and Approach. In the SGT model, during the graph data construction process, we map the features of nodes to a lower-dimensional space. However, there is currently no research indicating whether the feature dimension of nodes affects the model performance. On one hand, a higher node dimension contains more information, which can positively guide the model. However, excessively high feature dimensions may introduce redundant features, thereby reducing the model's performance. Therefore, this experiment discusses the impact of node feature dimensions on the model's performance. We select five feature dimensions ranging from low to high (16, 32, 64, 128, 256) to observe the performance variations of the SGT model across three datasets, and all the graphs have 100 nodes. We report the model's AUC, F1, Balance, Precision, and MCC results. The implementation details of the experiments align with RQ1.

Results. All the experiment results are shown in Fig. 11, Tables 12, 13, and 14. Fig. 11 is the model performance trend of an error band plot graph on different node feature dimensions, we show the model performance using five evaluation metrics. Tables 12, 13, and 14 are the average predictive performance of the SGT model under different node feature dimensions in the Linux, MySQL, and NetBSD datasets, separately. In each method, we specified the SGT for comparison along with the node feature dimension we employed, which is the 'SGT + feature dimension'.

Table 12
Average predictive performance of the SGT model under different feature dimensions in the Linux dataset.

Methods	AUC	F1	Balance	Precision	MCC
SGT-16	0.969	0.651	0.957	0.513	0.689
SGT-32	0.970	0.663	0.957	0.563	0.701
SGT-64	0.986	0.705	0.981	0.576	0.747
SGT-128	0.987	0.706	0.982	0.561	0.734
SGT-256	**0.996**	**0.726**	**0.994**	**0.579**	**0.754**

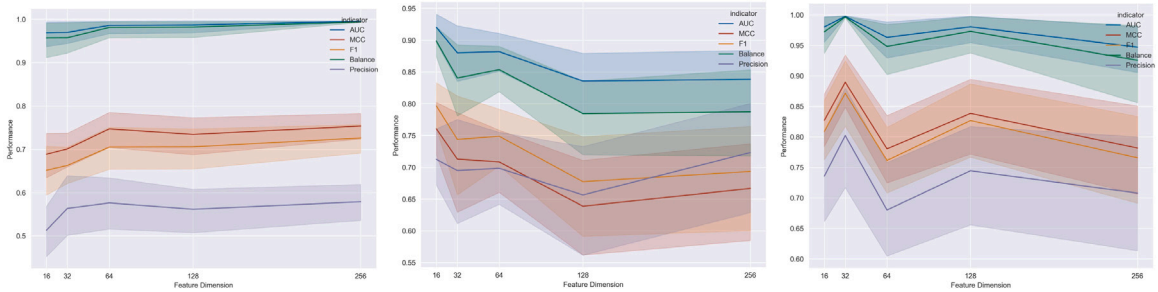
Table 13
Average predictive performance of the SGT model under different feature dimensions in the MySQL dataset.

Methods	AUC	F1	Balance	Precision	MCC
SGT-16	**0.919**	**0.796**	**0.878**	0.714	**0.760**
SGT-32	0.881	0.736	0.842	0.673	0.704
SGT-64	0.885	0.754	0.856	0.705	0.716
SGT-128	0.836	0.678	0.784	0.659	0.641
SGT-256	0.839	0.694	0.787	**0.723**	0.667

From Fig. 11, we can observe that, on the Linux dataset, the AUC, F1, Balance, Precision, and MCC of the SGT model consistently increase with the node feature dimension increases, reaching the optimal values at a dimension of 256. We did not test dimensions larger than 256 because as the node dimension increases, the computational complexity of the model also increases, while the performance does not proportionally improve. On the MySQL dataset, as the node feature dimension increases, AUC, F1, Balance, and MCC decrease continuously, with the model achieving optimal performance at a dimension of 16. However, the Precision does not follow a consistent pattern, with the model achieving optimal performance at a dimension of 256. On the NetBSD dataset, we found that when the node feature dimension is 32, the SGT model can achieve optimal AUC and Balance, indicating a high classification ability. When the node feature dimension is 128, the SGT model can achieve optimal F1, Precision, and MCC, indicating high prediction accuracy.

In terms of the width of the error bands across the three datasets, it is evident that the performance of the SGT model is most stable on Linux, followed by NetBSD, and poorest on MySQL. This observation suggests that the SGT model exhibits superior generalization capabilities on the Linux dataset while demonstrating comparatively weaker generalization on the MySQL dataset. The model's prediction performance, though satisfactory across all three datasets, may be adversely impacted by the limited volume of training data available for the MySQL dataset. Notably, MySQL possesses the smallest dataset among the three, which can potentially result in the undertraining of the model and consequently impair its performance on novel data. Moreover, the lack of diversity and representativeness in the MySQL dataset could contribute to the observed deficiency in generalization. When training data fails to comprehensively encompass the breadth of the problem space, the model may not effectively capture essential data features and patterns, adversely affecting its performance on new data. Thus, enhancing both the volume and diversity of the MySQL dataset could be conducive to boosting the model's generalization capacity and overall performance on this specific dataset. To accurately assess the performance differences of the models, we report the average results of various indicators under different dimensions on each dataset, as shown in Tables 12, 13, and 14.

From Table 12, we can observe that on the Linux dataset, optimal performance is achieved when the node feature dimension is set to 256. Specifically, the model achieves an AUC of 0.996, F1 score of 0.726, Balance of 0.994, Precision of 0.579, and MCC of 0.754. On the



(a) Performance on Linux (b) Performance on MySQL (c) Performance on NetBSD

Fig. 11. The performance trend (in terms of AUC, F1, Balance, Precision, and MCC) on different node feature dimensions.

Table 14
Average predictive performance of the SGT model under different feature dimensions in the NetBSD dataset.

Methods	AUC	F1	Balance	Precision	MCC
SGT-16	0.980	0.809	0.972	0.736	0.827
SGT-32	**0.991**	0.6	**0.987**	0.429	0.813
SGT-64	0.963	0.761	0.948	0.680	0.780
SGT-128	0.980	**0.827**	0.973	**0.744**	**0.838**
SGT-256	0.947	0.766	0.925	0.708	0.782

MySQL dataset, detailed in Table 13, optimal performance is observed with a node feature dimension of 16, resulting in an AUC of 0.919, F1 score of 0.796, Balance of 0.878, and MCC of 0.760. Notably, at a node feature dimension of 256, the model achieves an optimal MCC of 0.723. Likewise, the NetBSD dataset, outlined in Table 14, demonstrates optimal performance with a node feature dimension of 32, yielding an AUC of 0.991 and Balance of 0.987. Moreover, at a node feature dimension of 128, the model achieves optimal performance in terms of F1 score (0.827), Precision (0.744), and MCC (0.838).

In terms of the results on Linux and NetBSD, it is notable that the Precision for SGT are 0.579 and 0.744, respectively. This indicates that class imbalance can significantly impact the model's prediction accuracy. Additionally, when contrasting the AUC of SGT across MySQL, Linux, and NetBSD datasets, the AUC is 0.919, 0.996, and 0.991, respectively. This observation underscores the robust classification capacity of the SGT model when applied to the Linux dataset, demonstrating superior discrimination between positive and negative instances. However, the diminished performance on the MySQL dataset, attributable to its smaller size, is evident in the comparatively lower AUC. Conversely, despite the high class imbalance problem on the NetBSD dataset, the model still achieves a commendable AUC, indicative of its adeptness in classification tasks within this dataset. Hence, it is posited that dataset size plays a pivotal role in ensuring model stability.

Answer to RQ5: On the Linux dataset, the best graph node feature dimension is 256. On the MySQL dataset, when the feature dimension is 16, the SGT model achieves optimal AUC, F1, Balance, and MCC. When the feature dimension is 256, the SGT model achieves optimal Precision. On the NetBSD dataset, when the feature dimension is 32, the SGT model achieves optimal AUC and Balance. When the feature dimension is 128, the SGT model achieves optimal F1, Precision, and MCC.

6. Threats to the validity

In this section, we discuss the threats to the validity of our study.

(1) In our paper, the Linux, MySQL, and NetBSD dataset are chosen for the experiment, because these three datasets are widely discussed in the ARBP field and contain abundant ARBs (Liu et al., 2020; Zhou et al., 2022; Qin et al., 2023; Zhang et al., 2023; Chouhan and Rathore, 2021; Xu et al., 2020; Li et al., 2021; Kumar and Sureka, 2018), we could not affirm that SGT still performs the best on other ARB datasets. Therefore, we will verify the generality of our method on more datasets in the future.

(2) In our experiment, when discussing the performance differences between traditional metric features and deep semantic features, we utilized the original 52-dimensional and 82-dimensional feature sets as feature inputs, without considering any feature selection methods to mitigate the interference of redundant features on the model.

(3) Randomness in experiments may affect the performance of the SGT model. In our experiment, we apply random oversampling, we cannot guarantee the same result in each run, because ROS copy ARB files randomly. Additionally, The initialization parameters for each training of the model are randomly generated, which may also introduce certain errors in the model's prediction accuracy. Therefore, we run the methods 30 times, and then obtain the average value of the 30 results as the final result on each dataset.

(4) This study faces potential threats to internal validity arising from implementation errors during experimentation and the quality of the datasets utilized. To address these concerns, we employed the standard implementation provided by the Python 3.7 platform for constructing our classification models, thus minimizing the possibility of experiment-related mistakes. Additionally, to ensure consistency with prior research, we employed the same datasets as those used in Liu et al. (2020), Zhou et al. (2022) and Qin et al. (2023). However, it is important to acknowledge that the datasets themselves may contain quality issues that could potentially impact our results. Furthermore, our data processing approach is different from the other method (Liu et al., 2020; Zhou et al., 2022; Qin et al., 2023), which can result in the loss of some invalid files, leading to inconsistencies in the final number of files.

7. Related work

7.1. Manually design-based software metrics extraction method

Manually design-based software metrics extraction method analyzes the source code by using tools such as clang (Clang, 2016) or understood and so on to generate the metrics to measure the source code quality. Researchers design these handcraft metrics for different requirements. For instance, metrics to measure code size and intrinsic complexity, such as line of code metric, operator and Halstead metric, McCabe cyclomatic complexity metric, CK metric, and so on. Metrics to measure software development processes, such as code churn, micro-interactions, network features based on dependency graphs, and so on. Before the rise of deep learning, researchers leveraged these features extracted from project repositories to train classifiers using machine learning to predict the Bugs in new files. For example, Kaur and Kaur (2022b) adopted McCabe cyclomatic complexity and Program

Size features to analyze ARBs in cloud-oriented software. Chouhan and Rathore (2021) discussed the class imbalance issue in MySQL and Linux datasets using an 82-dimensional feature set containing four types of metrics: Program size-related metrics, Cyclomatic complexity, Halstead metrics, and aging-related metrics. Xu et al. (2020) applied the 52-dimension feature set to solve the cross-project ARBP task, and Qin et al. (2020) also adopted this feature set to discuss how normalization methods, kernel functions, and machine learning classifiers affect the transfer Learning based cross-project ARBP model. Many outstanding works, such as Kaur and Kaur (2022a,c), Li et al. (2021) and Khanna et al. (2021), are based on existing feature sets for ARBP discussion.

Furthermore, researchers proposed some metrics highly correlated with software aging. For example, Cotroneo et al. (2013) introduced six aging-related metrics, which are AllocOps, DeallocOps, DerefSet, DerefUse, UniqueDerefSet, UniqueDerefUse and integrated them with common software complexity metrics (program size metrics, Halstead metrics, McCabe complexity metrics) to construct ARBP models. These metrics focused on memory management and data structure, which center on primitives related to memory allocation and filesystem access, potentially increasing the likelihood of resource leakage. Qin et al. (2023) proposed network features based on dependency graphs to model the flow of aging-related information in the software. They think that most of the research commonly employs 52-dimensional metrics, with only a small subset of six metrics tailored specifically for ARBs. These metrics predominantly aim to capture the general complexity of source code, lacking the capacity to differentiate ARBs from general software bugs effectively. Furthermore, the metrics utilized primarily focus on single files, neglecting the potential interactions related to aging across multiple files. However, in real software systems, some research (Zhou et al., 2022; Liu et al., 2020; Bryan and Moriano, 2023) indicated that existing handcraft metrics often fail to distinguish codes with different semantics. Specifically, codes with different semantics can possess similar or even identical values for the above metrics.

7.2. Machine learning-based deep semantic feature learning method

Machine learning-based deep semantic feature learning method is proposed to compensate for the shortcomings of the manually design-based software metrics. This method learns the semantic and syntax features from the source code, which can better understand the logical structure. Commonly used feature extraction methods in the software defect prediction field include the token sequence-based feature extraction method and graph-based feature (AST trees, code property graphs, data dependency graphs, control flow graphs, etc.) extraction method.

The token sequence-based feature extraction method usually extracts token sequences from codes' ASTs for deep feature learning. It can efficiently bridge the gap between the programs' semantic information and features used for defect prediction. For example, Liu et al. (2020) proposed a token sequence-based feature extraction method. They utilized the Clang tool (Clang, 2016) to extract AST features from the source code, and then token sequences were parsed and combined with six handcraft ARB metrics mentioned in Cotroneo et al. (2013) as the final features. Finally, features were converted as vectors and fed into a convolution neural network for the ARBP task. Zhou et al. (2022) followed the idea of Liu et al. (2020), but utilized the token-level features with an improved bidirectional LSTM based on attention mechanisms as the feature learning model for the ARBP task. However, the token sequence-based feature extraction method fails to represent control relationships and data relationships among code lines and the practice of assigning an overall semantic feature to a code file leads to the loss of semantic features for specific code elements.

Graph-based feature extraction method converts the source code as the graph representation, such as control flow graph, data flow graph, code property graph, etc. For example, Bryan and Moriano (2023) constructed the contribution graphs related to the developers and source files for a just-in-time defect prediction task. Specifically, they utilized contribution graphs to model developers' changes to software files, employing this framework for edge classification. Zhou et al. (2022) constructed ASTs from source code, encoding symbolic tokens into numerical vectors for CNN input, and utilized a Class Dependency Network with a Graph Convolutional Network to learn structural information. The learned features combined with traditional hand-crafted features, improved defect prediction accuracy. Abdu et al. (2024) supposed that single representation is still limited for predicting defects that call multiple functions and have a high probability of false positives. They proposed a defect prediction model based on multiple source code representations, which is a deep hierarchical convolutional neural network (DH-CNN). DH-CNN combined the syntax features extracted from abstract syntax trees and the semantic-graph features extracted from the control flow graph and data dependence graph to model the semantic-level classification model. Tang et al. (2022) proposed a method, DP-GCN, which utilized Graph Convolutional Network to simultaneously capture semantic and structural information from Abstract Syntax Trees (ASTs) of source code, addressing the limitations of DBN, CNN, and LSTM in preserving AST structure. DP-GCN combined semantic and structural features extracted from ASTs with handcrafted metrics to train a Logistic Regression classifier for defect prediction, enhancing prediction accuracy. Graph-based feature extraction methods can compensate for the deficiencies of the model in the token sequence-based feature extraction method. However, in large-scale development projects, the high complexity of the original graph can render the model ineffective or even lead to model failure.

In addition, there are also some hybrid models or empirical studies discussing the performance of deep semantic feature models. For example, Zhu et al. (2022) treated the source code, requirements etc., as natural language text, and extracted features from these artifacts using embeddings generated through techniques such as Word2Vec (Church, 2017) to perform traceability link recovery tasks. Aung et al. (2022) introduced a refined approach for representing features of issue reports within bug triage models, in which the contexts of bug reports and code snippets are segregated into distinct tokens. Specifically, in the phase of code snippet feature extraction, an AST extractor is employed to parse each code snippet from an issue report and construct an AST path. Subsequently, these AST features are integrated with the natural language representations of the issue reports to facilitate the execution of bug triage tasks. Cheng et al. (2022) conducted a comparative analysis of vulnerability localization methods across various levels, such as method-level and slice-level, contrasting traditional static analysis techniques with machine learning-based deep semantic models. Their empirical study revealed substantial disparities between the capabilities of deep semantic models and static analysis models. They offered strategic recommendations to harness the strengths of both static and learning-based detectors. Specifically, they suggested developing methods that integrate the insights from traditional static analysis to enhance the informativeness of bug reports generated from the predictive and interpretative outcomes of deep learning models.

Based on our current investigation, token-based methods are commonly employed in the SDP, while graph-based methods are mostly used in vulnerability detection. Although other methods treat source code and similar data as natural language, they do not discuss how to efficiently extract features in large-scale complex projects. In the ARBP domain, current discussions on deep learning methods are primarily limited to token-based approaches, graph-level method is a niche area. We think the difference in ARBP compared to the aforementioned domains is that there is a notable complexity difference in source code systems. Vulnerability detection analyzes code segments for injected vulnerabilities, leading to simpler graph structures. Conversely, ARBP tasks involve complex logical code in source files, extending to tens of thousands of lines. Applying methods from vulnerability analysis may yield excessively large graphs, escalating computational complexity and impeding effective model training. Therefore, effective feature extraction methods are needed to enable the model to learn code features, particularly features related to ARBs.

8. Conclusion and future work

In this study, we propose a novel method for predicting software aging-related bugs, termed the SGT model, based on improved Graph-Transformer deep semantic feature learning. The SGT model effectively utilizes code property graphs to represent source code, thereby preserving semantic information. To address the complexity of graph structures in complex code systems, we introduce a sub-graph sampling method based on node degree. This approach reduces the structural complexity of the original graph by retaining only key nodes and branches with high utility. Additionally, we employ random oversampling to increase the number of ARB-prone instances, mitigating the risk of model ineffectiveness during training. Experimental evaluations conducted on three widely used open-source projects demonstrate that the SGT model significantly enhances predictive performance, particularly the F1 score. For example, on the Linux dataset, notable improvements were observed in 12.3%. On the MySQL dataset, improvements were observed in 6.8%.

In our future work, we aim to enhance our model in the following areas: (1) Improving the sub-graph sampling method. The graph sampling method based on node degree deletes a large number of nodes and edges during the sampling process, which disrupts the original structural features of the large graph and introduces significant uncertainty for subsequent data analysis. In future work, we plan to design feature-driven graph sampling methods (Wang et al., 2024) that selectively preserve attributes such as nodes and edges from the original network graph. The goal is to simplify the graph structure effectively while maintaining the essential features, to enable reasonable utilization of graph sampling methods for different datasets.

(2) Improving class imbalance resampling method. Random oversampling we used increases the number of minority class instances by replicating existing ones, which can lead to overfitting, especially when the dataset is small. In addition, randomly duplicating instances without considering the underlying data distribution may exacerbate the presence of noise in the dataset, making it challenging for the model to discern meaningful patterns. We should consider more reasonable graph-based rebalancing sampling methods to address the highly imbalanced nature of the dataset, such as Graph-SMOTE or imbalanced graph feature learning method (Wang et al., 2022). In addition, we also plan to compare traditional sampling methods (Tong et al., 2016; Zhao et al., 2020; Lin et al., 2020) to explore the impact of different sub-graph sampling methods on the performance of our model.

CRediT authorship contribution statement

Chen Zhang: Writing – review & editing, Writing – original draft. **Jianwen Xiang:** Supervision. **Rui Hao:** Writing – review & editing. **Wenhua Hu:** Writing – review & editing. **Domenico Cotroneo:** Writing – review & editing. **Roberto Natella:** Writing – review & editing. **Roberto Pietrantuono:** Writing – review & editing.

Declaration of competing interest

The authors of this paper certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honorarium; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

Data availability

Data will be made available on request.

Acknowledgments

This work is partially supported by the National Key Research and Development Program (Grant No. 2022YFB3104001), the National Key Research and Development Program (Grant No. 62202350), the Key Research and Development Program of Hubei Province (Grant No. 2022BAA050) and the Natural Science Foundation of Chongqing, China (Grant No. cstc2021jcyj-msxmX1146).

References

Abdu, A., Zhai, Z., Abdo, H.A., Algabri, R., 2024. Software defect prediction based on deep representation learning of source code from contextual syntax and semantic graph. IEEE Trans. Reliab..

Ali, H., Salleh, M.M., Saedudin, R., Hussain, K., Mushtaq, M.F., 2019. Imbalance class problems in data mining: A review. Indones. J. Electr. Eng. Comput. Sci. 14 (3), 1560–1571.

Allen, F.E., 1970. Control flow analysis. ACM Sigplan Not. 5 (7), 1–19.

Aung, T.W.W., Wan, Y., Huo, H., Sui, Y., 2022. Multi-triage: A multi-task learning framework for bug triage. J. Syst. Softw. 184, 111133.

Baah, G.K., Podgurski, A., Harrold, M.J., 2008. The probabilistic program dependence graph and its application to fault diagnosis. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 189–200.

Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, pp. 368–377.

Bhatia, V., Rani, R., 2017. An efficient algorithm for sampling of a single large graph. In: 2017 Tenth International Conference on Contemporary Computing. IC3, IEEE, pp. 1–6.

Bhatia, N., et al., 2010. Survey of nearest neighbor techniques. arXiv preprint arXiv:1007.0085.

Bhavsar, H., Panchal, M.H., 2012. A review on support vector machine for data classification. Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET) 1 (10), 185–189.

Bryan, J., Moriano, P., 2023. Graph-based machine learning improves just-in-time defect prediction. PLoS ONE 18 (4), e0284077.

Chen, D., O'Bray, L., Borgwardt, K., 2022. Structure-aware transformer for graph representation learning. In: International Conference on Machine Learning. PMLR, pp. 3469–3489.

Cheng, X., Nie, X., Li, N., Wang, H., Zheng, Z., Sui, Y., 2022. How about bug-triggering paths?-understanding and characterizing learning-based vulnerability detectors. IEEE Trans. Dependable Secure Comput..

Chouhan, S.S., Rathore, S.S., 2021. Generative adversarial networks-based imbalance learning in software aging-related bug prediction. IEEE Trans. Reliab. 70 (2), 626–642.

Church, K.W., 2017. Word2vec. Nat. Lang. Eng. 23 (1), 155–162.

Clang, L., 2016. AC language family frontend for llvm.

Cotroneo, D., De Simone, L., Natella, R., Pietrantuono, R., Russo, S., 2022. Software micro-rejuvenation for android mobile systems. J. Syst. Softw. 186, 111181.

Cotroneo, D., Natella, R., Pietrantuono, R., 2013. Predicting aging-related bugs using software complexity metrics. Perform. Eval. 70 (3), 163–178.

Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S., 2010. Software aging analysis of the linux operating system. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. IEEE, pp. 71–80.

da Costa, J.T., Matos, R.d.S., de Araujo, J.C., Maciel, P.R., 2021. Systematic mapping of literature on software aging and rejuvenation research trends. In: 2021 Annual Reliability and Maintainability Symposium. RAMS, IEEE, pp. 1–6.

Feng, S., Keung, J., Liu, J., Xiao, Y., Yu, X., Zhang, M., 2021. ROCT: Radius-based class overlap cleaning technique to alleviate the class overlap problem in software defect prediction. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, pp. 228–237.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.

Gascon, H., Yamaguchi, F., Arp, D., Rieck, K., 2013. Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security. pp. 45–54.

Ghanavati, M., Costa, D., Seboek, J., Lo, D., Andrzejak, A., 2020. Memory and resource leak defects and their repairs in java projects. Empir. Softw. Eng. 25, 678–718.

Gong, L., Zhang, H., Zhang, J., Wei, M., Huang, Z., 2022. A comprehensive investigation of the impact of class overlap on software defect prediction. IEEE Trans. Softw. Eng. 49 (4), 2440–2458.

Hammer, C., Snelting, G., 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. 8 (6), 399–422.

Huang, Y., Kintala, C., 1993. Software implemented fault tolerance: Technologies and experience. In: FTCS. Vol. 23, IEEE Computer Society Press, pp. 2–9.

Kalantari, K.R., Ebrahimnejad, A., Motameni, H., 2020. Dynamic software rejuvenation in web services: a whale optimizationalgorithm-based approach. Turk. J. Electr. Eng. Comput. Sci. 28 (2), 890–903.

Kaur, A., Kaur, H., 2022a. Prediction of aging-related bugs using software code metrics. In: Intelligent System Algorithms and Applications in Science and Technology. Apple Academic Press, pp. 171–182.

Kaur, H., Kaur, A., 2022b. An empirical study of aging related bug prediction using cross project in cloud oriented software. Informatica (Ljubl.) 46 (8).

Kaur, H., Kaur, A., 2022c. Predicting aging related bugs with automated feature selection techniques in cloud oriented softwares. In: International Conference on Advances and Applications of Artificial Intelligence and Machine Learning. Springer, pp. 217–231.

Khanna, M., Aggarwal, M., Singhal, N., 2021. Empirical analysis of artificial immune system algorithms for aging related bug prediction. In: 2021 7th International Conference on Advanced Computing and Communication Systems. ICACCS, Vol. 1, IEEE, pp. 692–697.

Kotsiantis, S.B., Zaharakis, I., Pintelas, P., et al., 2007. Supervised machine learning: A review of classification techniques. In: Emerging Artificial Intelligence Applications in Computer Engineering. Vol. 160, Amsterdam, pp. 3–24.

Kumar, L., Singh, V., Murthy, L.B., Misra, S., Krishna, A., 2023. An empirical framework for software aging-related bug prediction using weighted extreme learning machine. Ann. Comput. Sci. Inf. Syst. 37, 187–194.

Kumar, L., Sureka, A., 2018. Feature selection techniques to counter class imbalance problem for aging related bug prediction: aging related bug prediction. In: Proceedings of the 11th Innovations in Software Engineering Conference. pp. 1–11.

Li, D., Liang, M., Xu, B., Yu, X., Zhou, J., Xiang, J., 2021. A cross-project aging-related bug prediction approach based on joint probability domain adaptation and k-means SMOTE. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion. QRS-C, IEEE, pp. 350–358.

Lin, M., Li, W., Lu, S., 2020. Balanced influence maximization in attributed social network based on sampling. In: Proceedings of the 13th International Conference on Web Search and Data Mining. pp. 375–383.

Liu, Q., Xiang, J., Xu, B., Zhao, D., Hu, W., Wang, J., 2020. Aging-related bugs prediction via convolutional neural network. In: 2020 7th International Conference on Dependable Systems and their Applications. DSA, IEEE, pp. 90–98.

Liu, Z., Zheng, Y., Du, X., Hu, Z., Ding, W., Miao, Y., Zheng, Z., 2022. Taxonomy of aging-related bugs in deep learning libraries. In: 2022 IEEE 33rd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 423–434.

Liu, Z., Zhou, J., 2022. Introduction to Graph Neural Networks. Springer Nature.

Myles, A.J., Feudale, R.N., Liu, Y., Woody, N.A., Brown, S.D., 2004. An introduction to decision tree modeling. J. Chemom.: J. Chemom. Soc. 18 (6), 275–285.

Neamtiu, I., Foster, J.S., Hicks, M., 2005. Understanding source code evolution using abstract syntax tree matching. In: Proceedings of the 2005 International Workshop on Mining Software Repositories. pp. 1–5.

Palmer, A., 2022. Amazon Web Services Outage Brings Some Delivery Operations to a Standstill. CNBC, Retrieved 12 February.

Pandey, S.K., Tripathi, A.K., 2021. An empirical study toward dealing with noise and class imbalance issues in software defect prediction. Soft Comput. 25 (21), 13465–13492.

Qin, F., Wan, X., Yin, B., 2020. An empirical study of factors affecting cross-project aging-related bug prediction with TLAP. Softw. Qual. J. 28, 107–134.

Qin, F., Zheng, Z., Wan, X., Liu, Z., Shi, Z., 2023. Predicting aging-related bugs using network analysis on aging-related dependency networks. IEEE Trans. Emerg. Top. Comput..

Robnik-Šikonja, M., Kononenko, I., 2003. Theoretical and empirical analysis of relieff and rrelieff. Mach. Learn. 53, 23–69.

Salamanos, N., Voudigari, E., Yannakoudakis, E.J., 2017. Identifying influential spreaders by graph sampling. In: Complex Networks & their Applications V: Proceedings of the 5th International Workshop on Complex Networks and their Applications. COMPLEX NETWORKS 2016, Springer, pp. 111–122.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2008. The graph neural network model. IEEE Trans. Neural Netw. 20 (1), 61–80.

Sparks, S., Embleton, S., Cunningham, R., Zou, C., 2007. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual Computer Security Applications Conference. ACSAC 2007, IEEE, pp. 477–486.

Tang, L., Tao, C., Guo, H., Zhang, J., 2022. Software defect prediction via GCN based on structural and context information. In: 2022 9th International Conference on Dependable Systems and their Applications. DSA, IEEE, pp. 310–319.

Tong, C., Lian, Y., Niu, J., Xie, Z., Zhang, Y., 2016. A novel green algorithm for sampling complex networks. J. Netw. Comput. Appl. 59, 55–62.

Wang, S., Liu, T., Nam, J., Tan, L., 2018. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. 46 (12), 1267–1293.

Wang, X., Shi, J.-H., Zou, J.-J., Shen, L.-Z., Lan, Z., Fang, Y., Xie, W.-B., 2024. Supports estimation via graph sampling. Expert Syst. Appl. 240, 122554.

Wang, Y., Zhao, Y., Shah, N., Derr, T., 2022. Imbalanced graph classification via graph-of-graph neural networks. In: Proceedings of the 31st ACM International Conference on Information & Knowledge Management. pp. 2067–2076.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., et al., 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315.

Xia, F., Sun, K., Yu, S., Aziz, A., Wan, L., Pan, S., Liu, H., 2021. Graph learning: A survey. IEEE Trans. Artif. Intell. 2 (2), 109–127.

Xu, J., Ai, J., Liu, J., Shi, T., 2022. ACGDP: An augmented code graph-based system for software defect prediction. IEEE Trans. Reliab. 71 (2), 850–864.

Xu, B., Zhao, D., Jia, K., Zhou, J., Tian, J., Xiang, J., 2020. Cross-project aging-related bug prediction based on joint distribution adaptation and improved subclass discriminant analysis. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 325–334.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Zafeiriou, S., Bronstein, M., Cohen, T., Vinyals, O., Song, L., Leskovec, J., Lio, P., Bruna, J., Gori, M., 2022. Guest editorial: Non-euclidean machine learning. IEEE Trans. Pattern Anal. Mach. Intell. 44 (2), 723–726.

Zhang, C., Feng, S., Xie, W., Zhao, D., Xiang, J., Pietrantuono, R., Natella, R., Cotroneo, D., 2023. IFCM: An improved fuzzy C-means clustering method to handle class overlap on aging-related software bug prediction. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 590–600.

Zhao, Y., Jiang, H., Qin, Y., Xie, H., Wu, Y., Liu, S., Zhou, Z., Xia, J., Zhou, F., et al., 2020. Preserving minority structures in graph sampling. IEEE Trans. Vis. Comput. Graphics 27 (2), 1698–1708.

Zhao, K., Xu, Z., Yan, M., Zhang, T., Yang, D., Li, W., 2021. A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models. Inf. Softw. Technol. 139, 106652.

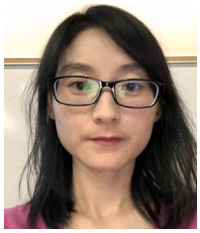
Zhou, Y., Zhang, C., Jia, K., Zhao, D., Xiang, J., 2022. A software aging-related bug prediction framework based on deep learning and weakly supervised oversampling. In: 2022 IEEE International Symposium on Software Reliability Engineering Workshops. ISSREW, IEEE, pp. 185–192.

Zhu, J., Xiao, G., Zheng, Z., Sui, Y., 2022. Enhancing traceability link recovery with unlabeled data. In: 2022 IEEE 33rd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 446–457.

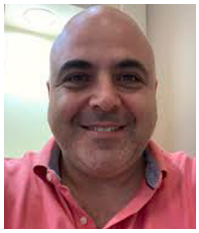
Chen Zhang is currently pursuing the doctoral degree with the Wuhan University of Technology, Wuhan, China. Her current research interests include software aging and software defect prediction.

Jianwen Xiang received his B.S. and M.S. degrees in Computer Science from Wuhan University in 1997 and 2000, respectively. He received his Ph.D. in Computer Science from Japan Advanced Institute of Science and Technology (JAIST) in 2005. He is currently a full professor of the School of Computer Science and Artificial Intelligence of Wuhan University of Technology. His research interests include dependable computing and software engineering.

Rui Hao received the Ph.D. degree from Nanjing University (NJU), Nanjing, China in 2023. She is currently a Post-Doctoral Researcher with the School of Computer Science and Artificial Intelligence, Wuhan University of Technology, China. Her research interests include software quality and software security.



Wenhua Hu is an associate professor of School of Computer Science and Artificial Intelligence at the Wuhan University of Technology. She earned her PhD from the University of Alabama. Her research interests include empirical software engineering, requirements engineering, software quality. Contact her at whu10@whut.edu.cn.



Domenico Cotroneo is a full professor at the University of Naples Federico II, Italy. His research interests include software fault injection, dependability assessment, and field-based measurement techniques.

Roberto Natella PhD, is associate professor at the Federico II University of Naples. His research interests are in the field of software security and dependability. He authored more than 100 peer-review papers. The main recurring theme of his research activity is the experimental injection of faults, attacks, and stressful conditions. In 2022, he received the DSN 2022 Rising Star in Dependability Award from the IEEE TCFT and IFIP WG 10.4.

Roberto Pietrantuono received his B.S. and M.S. degrees in Computer Engineering from the University of Naples Federico II in 2003 and 2006, respectively. He received his Ph.D. in Computer and Automation Engineering from the same university in 2009. He is currently an associate professor of the Electrical Engineering and Information Technology Department at the University of Naples Federico II. His research interests include software engineering, dependability, and artificial intelligence.