DSLA: End-to-End Feature Fusion for Superior Fault Localization of Software

Qun Xiao[1,2], Ming Zhou[3], Jiaqian Peng[1,2], Shouguo Yang[4],
Zhanwei Song[*,1,2], Zhiqiang Shi[*,1,2] Zhi Li[1,2], Limin Sun[1,2]
[1]*Institute of Information Engineering, CAS,* Beijing, China
[2]*School of Cyber Security, University of Chinese Academy of Sciences,* Beijing, China
[3]*School of Cyber Science and Engineering, Nanjing University of Science and Technology,* Nanjing, China
[4]*Zhongguancun Laboratory,* Beijing, China
{songzhanwei,shizhiqiang}@iie.ac.cn

***Abstract*—Current fault localization methods struggle to prioritize true faulty lines due to coverage ambiguities in dynamic analysis and missing semantic dependencies in static analysis. Existing feature-concatenation strategies fail to produce sufficiently discriminative suspiciousness scores for Top-N recall. To address this, we propose DSLA, which constructs a line-level property graph integrating dynamic coverage with static code semantics, enabling fine-grained code representation. DSLA performs deep feature fusion via a joint objective combining MSE, t-SNE constraints, and discriminative losses, explicitly enforcing geometric separability to prevent model collapse. On the Defects4J benchmark, DSLA improves Top-1 recall by 1.3× over state-of-the-art techniques, and on the C-language Codeflaws dataset, it achieves a 6.1× improvement, demonstrating its effectiveness in precise faulty line localization.**

***Index Terms*—software fault localization, feature fusion, code semantics, code line coverage, multiple objective optimization.**

I. INTRODUCTION

As software systems and their supply chains continue to grow in complexity, software bugs have become an unavoidable challenge [1]. The effectiveness of fault localization and the subsequent bug-fixing process is crucial to ensuring software quality [2]. Fault localization seeks to identify suspicious program elements at various levels of granularity, ranging from broader entities such as files [3], embedded devices [4], [5], and high-level program units (e.g., modules [6] or components [7]) to more detailed elements like functions [8], [9] and individual lines of code [10], [11]. This process reduces the manual debugging effort required by developers [12], [13].

A major challenge in fault localization arises from runtime failure, such as call stacks, which only reflect the execution context at the time of failure and do not directly pinpoint the root cause of a defect [14]. The actual fault may originate from earlier executions or incorrect parameter propagation, and many defects do not even result in crashes, further limiting the effectiveness of call stack-based localization [15]–[17].

Spectrum-based fault localization (SBFL) techniques attempt to address this by analyzing dynamic execution spectra and code coverage, ranking program elements using formulas such as Ochiai [18], DStar [19], and Tarantula [20]. However, SBFL's accuracy heavily depends on the quality of the test suite and often struggles to distinguish between code lines that have nearly identical or highly similar suspiciousness scores.

Recent advancements have integrated static semantic information with dynamic execution data through deep learning models. Examples include DeepFL [21], DeepRL4FL [22], TRANSFER-FL [10], and CMFL [23]. These approaches embed source code into vector representations and combine them with coverage matrices, SBFL scores, or historical fix data to predict fault locations. While they outperform traditional SBFL methods, they typically fail to fully capture structural dependencies, such as control-flow and data-flow relationships, and often perform static-dynamic feature fusion in separate stages, limiting their discriminative power.

Similarly, large language model (LLM)-based fault localization methods, such as LLMAO [11], focus on learning rich static semantic representations and can localize faults without dynamic execution data. However, these methods neglect the explicit structural dependencies between code lines, leading to incomplete modeling of program behavior.

To overcome the indistinguishability among highly suspicious code lines, we propose a unified framework that tightly integrates dynamic execution behaviors with static structural semantics. This integration is challenged by two key issues: (**C1**) the *granularity mismatch* between line-level dynamic traces and fine-grained static representations, and (**C2**) the *superficiality of existing fusion strategies*, which rely on naive concatenation. As a result, prior approaches lack a unified optimization objective that explicitly enforces geometric separability in the fused feature space.

To address these challenges, we model static semantics as graphs and align them with line-level fault localization by decomposing basic blocks into line-level nodes and reconstructing graph connectivity accordingly. Dynamic coverage is computed at the same granularity and fused with static features in a shared embedding space. We further project dynamic, static, and fused representations via PCA [24] and t-SNE [25], and jointly optimize cosine similarity and Euclidean distance to enhance both feature alignment and discriminative power.

Our contributions are summarized as follows:

- We propose **DSLA**, a unified dynamic–static fusion framework that integrates execution coverage with struc-

Zhanwei Song* and Zhiqiang Shi* are the corresponding authors.

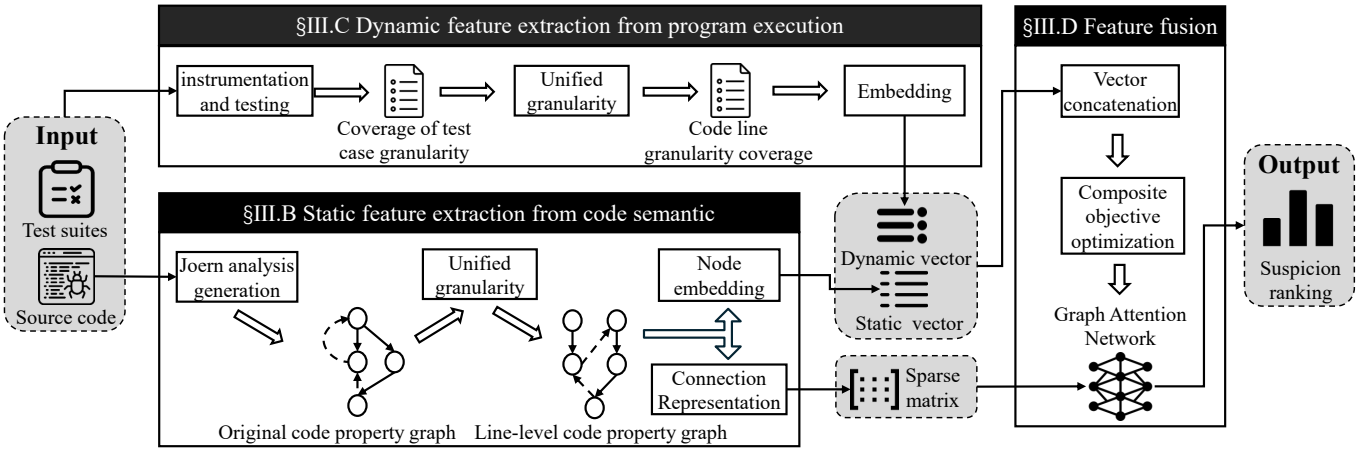

Fig. 1: The workflow of the DSLA framework

tural semantics to resolve indistinguishability in line-level fault localization.

- We introduce a line-level alignment and end-to-end collaborative optimization mechanism, leveraging a multi-objective graph attention network to enable deep, unified fusion beyond staged or concatenation-based methods.
- Extensive experiments on Defects4J and Codeflaws show that DSLA outperforms state-of-the-art approaches, achieving over **1.3×** improvement on Defects4J and an average **6.1×** gain across the top-10 defect types in Codeflaws.

The implementation and datasets are publicly available at https://anonymous.4open.science/r/dsla_afl_for_review-711D.

II. MOTIVATION

Automated fault localization is currently dominated by two paradigms: SBFL and static semantic analysis. Howevser, both exhibit complementary limitations regarding feature discriminability, preventing precise fault isolation.

On the one hand, SBFL methods transform test execution coverage into suspiciousness scores but treat code lines within the same basic block as homogeneous entities. This context-oblivious approach ignores the semantic differences between statements. As vividly exemplified by Mockito bug #29 in Fig. 2, the faulty statement (line 29) shares an identical Ochiai score (0.447) with its distinct context lines (27–28). Statistically, based solely on dynamic spectra, the root cause is **indistinguishable** from its execution context. On the other hand, while static analysis captures rich semantic information (e.g., identifying the potential null dereference at line 29), it inherently suffers from a lack of runtime focus. Without the guidance of failure-inducing executions, static features tend to flag all semantically risky patterns across the codebase equally, burying the actual fault amidst numerous benign "code smells" or latent risks that were never triggered.

To resolve this mutual indistinguishability, we propose DSLA, which leverages the **complementary synergy** of static and dynamic modalities. The fundamental rationale behind this

```
@@ -26,7 +26,7 @@ public boolean matches(Object actual) {
26      public void describeTo(Description description) {
27          description.appendText("same(");                  // SBFL=0.447
28          appendQuoting(description);                       // SBFL=0.447
29 -        description.appendText(wanted.toString());        // SBFL=0.447
29 +        description.appendText(wanted == null ? "null" : wanted.toString());
            ……                                                }
```

Fig. 2: Code snippet from Mockito bug #29 in the Java mocking framework. The defect causes a `NullPointerException` at line 29 due to method invocation on a potentially null object. Selected lines are annotated with Ochiai suspiciousness scores.

fusion is that a true fault possesses a unique fused signature: it must be simultaneously "executed during failure" and "semantically vulnerable." By synthesizing these dimensions, DSLA resolves the ambiguity observed in single-modal approaches. Specifically, in the Mockito example, although the faulty line 29 and the safe line 27 are dynamically identical due to shared execution paths, they are semantically distinct—the former involves a risky object dereference while the latter is a benign string append. Fusion allows the model to prioritize the semantically riskier statement within the executed block. Conversely, compared to other statically similar null-checks that remain dormant, line 29 is distinguished by its execution relevance. By mapping this synthesis of "execution context" and "semantic risk" into a unified latent space, DSLA constructs a fine-grained decision boundary that effectively isolates the active fault from both dynamically correlated safe lines and statically risky but irrelevant ones.

III. APPROACH

A. Overview

As depicted in Fig. 1, the DSLA framework comprises three sequential stages to address the core challenges of semantic alignment and feature fusion. First, to address **C1**, the system constructs a line-level CPG by parsing faulty programs with Joern and transforming them into graphs that establish a one-

Algorithm 1 Line-Level Graph Construction.

Input: Raw $\mathcal{N}, \mathcal{E}_{\text{raw}}, L_{\text{phys}}$; **Output:** $\mathcal{N}_{\text{line}}, \mathcal{E}_{\text{line}}$
procedure BUILDLINEGRAPH($\mathcal{N}, \mathcal{E}_{\text{raw}}$)
    Init $M_{\text{nl}}, M_{\text{id}}, \mathcal{N}_{\text{line}}, \mathcal{E}_{\text{line}} \leftarrow \emptyset$
    for $n \in \mathcal{N}$ **do** ▷ Step 1: Node Splitting
        Split n by \n into $\{n'\}$; add each n' to $M_{\text{nl}}[n'.line]$
    end for
    for ℓ, nodes $\in M_{\text{nl}}$ **do** ▷ Step 2: Aggregation
        $n^* \leftarrow \arg\max_{n \in \text{nodes}} \text{Score}(n)$; $\mathcal{N}_{\text{line}} \leftarrow \mathcal{N}_{\text{line}} \cup \{n^*\}$
        Map all IDs in nodes to $n^*.id$ in M_{id}
    end for
    for $(u, v, t) \in \mathcal{E}_{\text{raw}}$ **do** ▷ Step 3: Edge Re-establishment
        $u', v' \leftarrow M_{\text{id}}[u], M_{\text{id}}[v]$
        if $u' \neq \text{null} \wedge v' \neq \text{null} \wedge u' \neq v'$ **then** $\mathcal{E}_{\text{line}}$.add$(u', v', t)$
        end if
    end for
    return $\mathcal{N}_{\text{line}}, \mathcal{E}_{\text{line}}$
end procedure

to-one alignment between static semantic features and structural dependencies. Simultaneously, defective programs are instrumented using tools such as Gcov [26] and Cobertura [27] to extract high-dimensional dynamic execution feature vectors from coverage statistics. Subsequently, to address **C2**, these static and dynamic features are concatenated and processed via a multi-head fusion network, utilizing composite objective optimization to enhance both fusion quality and suspiciousness discrimination. Finally, a Graph Attention Network (GATConv) incorporates the line-level structural relations to predict suspiciousness scores and rank the target code lines accordingly.

B. Static feature extraction from code semantic

To align static analysis with the granularity of fault localization, we implement a graph transformation pipeline that converts fine-grained, statement-based CPGs into a unified line-level representation. As detailed in Algorithms 1, this unification proceeds through three sequential stages: node splitting, node aggregation, and edge re-establishment. Initially, raw CPG nodes generated by Joern [28] are parsed and split to strictly correspond to physical source code lines. Subsequently, multiple nodes mapping to the same line are aggregated into a single representative node that maximizes semantic coverage, after which topological edges are reconstructed to preserve valid control and data flow dependencies. This transformation significantly clarifies structural semantics.

For semantic representation, we leverage the pre-trained RoBERTa model [29] to encode the normalized code text of each line. By applying mean pooling over the token embeddings, we derive a fixed-dimensional static feature vector $\mathbf{h}_i^{\text{static}}$. Concurrently, the structural topology of the generated line-level CPG is encoded as a sparse adjacency matrix, serving as the structural input for the subsequent fusion module.

C. Dynamic feature extraction from program execution

Dynamic feature extraction characterizes runtime program behavior by instrumenting the source code and executing the test suite, as shown in Fig. 1. The resulting coverage reports are initially generated at the test-case level. To enable line-level fault localization, DSLA aggregates these reports into a unified-granularity representation that pivots execution data from test cases to code lines. As illustrated in Listing 1, each line is associated with three raw execution statistics: total execution count, number of passing tests, and number of failing tests (e.g., *Same.java_LINE_35*). These features are normalized using logarithmic transformation and Z-score standardization to compress value ranges and reduce variance. The normalized vectors are then mapped into a high-dimensional embedding space through a gated neural encoder, in which a learnable gate modulates the output of a base MLP to emphasize informative runtime patterns. The resulting representation, $\mathbf{h}_i^{\text{dyn}}$, serves as the dynamic feature embedding for subsequent fusion with static structural semantics.

Listing 1: Partial coverage of code line granularity example.

```
{
  "org/mockito/internal/matchers/Same.java_LINE_35":
       {
    "id":"org/mockito/internal/matchers/Same.
        java_LINE_35",
    "lineNumber": 35,
    "covered_by_tests": 1,
    "passed_tests": 0,
    "failed_tests": 1,
    "execution_count": 2
  }
}
```

D. Feature fusion and suspicion prediction

To synthesize complementary information from static and dynamic modalities, we employ an end-to-end multi-modal graph neural network, as illustrated in Fig. 3. First, static and dynamic features are normalized, concatenated, and projected via a multi-head linear layer to generate the fused latent representation $\mathbf{F}$ (Equation 1).

$$\mathbf{F} = \Big\Vert_{k=1}^{K} \phi\left([\text{LN}(\mathbf{X}_S) \oplus \text{LN}(\mathbf{X}_D)] \cdot \mathbf{W}_k\right) \tag{1}$$

Subsequently, $\mathbf{F}$ is processed by GATConv layers to capture structural dependencies. A residual connection adds the initial representation $\mathbf{F}$ to the GCN output, which is then mapped by a prediction layer to produce the final suspicion ranking.

Optimization is driven by a composite objective $\mathcal{L}$ (Equation 2) that balances task performance and feature space structure.

$$\mathcal{L} = \mathcal{L}_m + \mathcal{L}_d + \lambda_t \mathcal{L}_t \tag{2}$$

Here, $\mathcal{L}_m$ represents the primary MSE regression loss (Equation 3). $\mathcal{L}_d$ ensures the retention of dynamic behaviors by maximizing the mutual information between fused and dynamic features via contrastive learning (Equation 4). Furthermore, $\mathcal{L}_t$ explicitly enforces geometric separability between defective and non-defective nodes through a t-SNE-based constraint (Equations 5). The optimized features are finally propagated through GATConv layers to generate suspicion rankings.

$$\mathcal{L}_m = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{3}$$

$$\mathcal{L}_d = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\text{sim}(\mathbf{f}_i, \mathbf{x}_i^d))}{\sum_{j\neq i}\exp(\text{sim}(\mathbf{f}_i, \mathbf{x}_j^d))} \tag{4}$$

$$\begin{aligned}\mathcal{L}_t = &\sum_{i,j}\max(0, \delta_h - \|\mathbf{z}_i - \mathbf{z}_j\|)\cdot\mathbb{I}_{y_i\neq y_j} \\ &+\sum_{i,j}\max(0, \|\mathbf{z}_i - \mathbf{z}_j\| - \delta_l)\cdot\mathbb{I}_{y_i=y_j}\end{aligned} \tag{5}$$

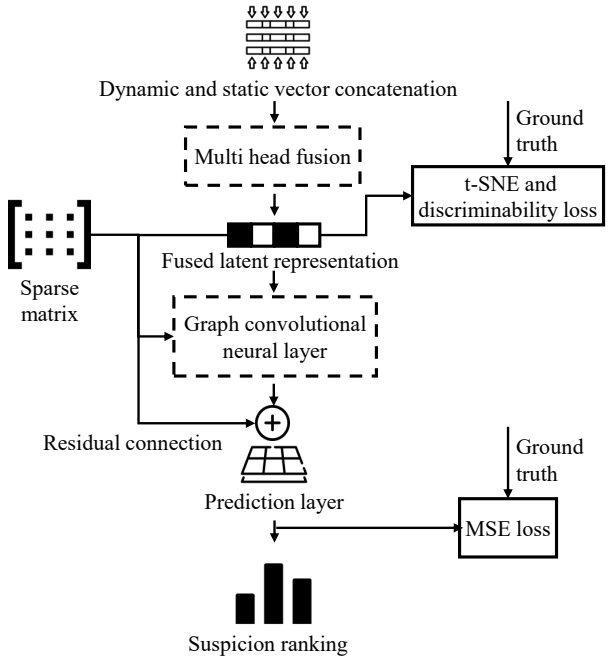

Fig. 3: Feature fusion-based suspiciousness ranking procedure.

IV. EVALUATION

- **RQ1**: How does DSLA perform against state-of-the-art fault localization techniques in terms of Top-*N* recall?
- **RQ2**: How robust is DSLA across programming languages with respect to Top-*N* recall?
- **RQ3**: How do DSLA's key components and composite loss function contribute to its overall Top-*N* recall?

A. Experimental Setup

All experiments were conducted under a unified and fixed configuration to ensure reproducibility and fair comparison across methods. The static encoder employs RoBERTa-base with a 768-dimensional embedding and a maximum input length of 512. Feature fusion is implemented using a four-head MultiHeadFusion module, followed by a three-layer GATConv network with two attention heads per layer, dropout rate 0.2, and residual connections. The prediction head consists of two linear layers with LeakyReLU activation (slope 0.1) and dropout. Models are trained using AdamW with a learning rate of 1×10^{-5}, batch size 16, and 50 epochs, together with edge dropping (rate 0.05), gradient clipping (max norm 1.0), and early stopping with a patience of five. Following prior work [10], [21], ten-fold cross-validation is adopted. All experiments are conducted on an Ubuntu 20.04 server equipped with 32 CPU cores, 125GB RAM, and four NVIDIA GTX 1080 Ti GPUs.

We evaluate DSLA on Defects4J [30] and Codeflaws [31] to assess both industrial applicability and cross-language generalization. Defects4J contains two versions of benchmark: v1.2.0 (6 projects) and the more diverse v2.0.0 (expanding to 17 projects), while Codeflaws contains 7,436 C programs across 40 defect types, we focus on the 10 most frequent types to ensure statistical reliability. Performance is measured using Recall@Top-N ($N = 1, 3, 5$), Mean First Rank (MFR), Mean Average Rank (MAR), and AUC, which collectively evaluate practical effectiveness, ranking quality, and overall discriminative capability [11], [32]. We compare DSLA against representative baselines, including the spectrum-based Ochiai method [18], the transfer-learning-based TRANSFER-FL [10], and the large language model approach LLMAO [11]. In addition, ablation studies are conducted with different code encoders (CodeGen-350M, CodeT5-large, and RoBERTa-base) and graph neural architectures (GATConv and GCNConv) to analyze the contribution of individual components.

B. RQ1. Comparison with the SOTA technique

We evaluate the effectiveness of DSLA against SOTA baselines on two versions of the Defects4J benchmark: v1.2.0 (395 faults) and the more diverse v2.0.0 (expanding to 17 projects). As presented in Table I, DSLA demonstrates superior performance on Defects4J v1.2.0, achieving a Top-1 recall of 58.2%, which significantly surpasses the best-performing baselines, CMFL (24.4%) and TRANSFER-FL (21.3%). Notably, regarding ranking quality metrics, DSLA reduces the Mean First Rank (MFR) to 8.73, an order of magnitude improvement over DeepFL (95.84) and Ochiai (183.78). This underscores the efficacy of our unified granular alignment and multi-modal fusion strategy. It is worth noting that the MFR and MAR metrics are omitted for LLMAO, since LLMAO operates on code fragments rather than entire files, it does not produce a global ranking of all code lines, making these metrics inapplicable.

On the diverse Defects4J v2.0.0 dataset, DSLA maintains robustness with a 41.6% Top-1 recall, surpassing the runner-up TRANSFER-FL (36.4%) and securing the best MFR (11.56). This confirms that DSLA's deep fusion effectively resolves the indistinguishability bottleneck that limits existing hybrid approaches like DeepFL.

C. RQ2. Cross-Language robustness analysis of DSLA

To assess cross-language generalizability, we compare DSLA with LLMAO on the C-language Codeflaws dataset, where traditional spectrum-based baselines are not directly applicable. As summarized in Table II, these subjects span four primary logical dimensions defined by the Codeflaws taxonomy: operand, operator, higher-order, and statement levels.

Fine-grained analysis by fault type further confirms this robustness in the Table III. For the ten most frequent fault categories DSLA consistently achieves TOP-1 recall rates

TABLE I: Comparison of Fault Localization Performance on Defects4J v1.2.0 and v2.0.0.

Feature Type	Technique	Defects4J v1.2.0					Defects4J v2.0.0				
		Top-1	**Top-3**	**Top-5**	**MFR**	**MAR**	**Top-1**	**Top-3**	**Top-5**	**MFR**	**MAR**
Dynamic	Ochiai [18]	4.8%	16.5%	25.1%	183.78	233.14	7.8%	19.4%	25.9%	136.24	224.57
Static	LLMAO [11]	22.3%	37.7%	46.3%	–	–	31.9%	41.2%	54.4%	–	–
Dynamic & Static	DeepFL [21]	14.1%	24.0%	34.0%	95.84	161.32	19.0%	41.3%	53.4%	84.37	137.61
	TRANSFER-FL [10]	21.3%	36.5%	43.3%	79.97	120.47	36.4%	**57.3%**	66.1%	13.59	16.72
	CMFL [23]*	24.4%	38.9%	47.3%	26.31	42.58	–	–	–	–	–
	DSLA (Ours)	**58.2%**	**64.6%**	**66.1%**	**8.73**	**13.21**	**41.6%**	55.7%	**67.3%**	**11.56**	**15.17**

* Results for CMFL are cited directly from its original paper as the source code is unavailable; no v2.0.0 data is available.
Note: MFR and MAR are not reported for LLMAO as it calculates suspicion scores on code fragments rather than comprehensive file-level rankings.

TABLE II: Taxonomy and examples of the top-10 defect classes from Codeflaws.

Taxonomy	ID	Defect Class Name	Example
Operand	DCCR	Replace constant with variable/constant	`- for(i=n+1;i<=9000;i++)` `+ for(i=n+1;i<=10000;i++)`
	DRVA	Replace a read variable with variable/constant	`- for(i=0;i<1;i++)` `+ for(i=0;i<m;i++)`
	DCCA	Modify array size	`- int x[100]` `+ int x[100000];`
Operator	ORRN	Replace relational operator	`- if(sum>n)` `+ if(sum>=n)`
	OILN	Insert && (tighten) or ‖ (loosen)	`- if(t%2==0)` `+ if(t%2==0 && t!=2)`
	OAIS	Insert/Delete arithmetic operator	`- max += days%2;` `+ max += (days%7)%2;`
Higher-order	HIMS	Insert multiple non-branch statements	`+ freopen("in.txt","r",stdin);` `+ freopen("out.txt","w",stdout);`
	HDMS	Delete multiple non-branch statements	`- freopen("in.txt","r",stdin);` `- freopen("out.txt","w",stdout);`

	HOTH	Other higher order defect classes	`- scanf("%s",h);`
			`+ for(i=0;i<71;i++)`
			`+ scanf("%c",&h[i]);`

Statement	STYP	Replace variable declaration type	`- int a;` `+ long a;`

above or close to 50%, with TOP-5 recall exceeding 90% in most cases, whereas LLMAO remains below 10% at TOP-1 for nearly all these categories. These results suggest that DSLA captures structural and semantic patterns that generalize across languages, while LLMAO is largely constrained by surface-level textual correlations. Although DSLA shows relatively weaker performance on a few rare fault types, its overall superiority demonstrates that it is a reliable and effective cross-language fault localization approach.

D. RQ3. The ablation studies of DSLA

To quantify the contribution of individual components, we analyze the results on Defects4J presented in Tables IV and V. Table IV validates the architectural superiority of the full DSLA configuration. Removing dynamic coverage precipitates a sharp decline in Top-1 recall (58.2% to 24.6%), confirming the necessity of execution traces. Furthermore, replacing GAT-Conv with GCNConv or substituting RoBERTa with generative models significantly degrades performance, highlighting the critical role of attention mechanisms and discriminative semantic embeddings. Regarding the optimization objective, Table V highlights a critical synergy among loss terms. While the baseline MSE yields a Top-1 recall of only 15.9%, integrating t-SNE or discriminability losses individually boosts performance to 34.1% and 26.3% respectively. However, the peak performance of 58.2% is only achieved when all terms are combined, demonstrating that enforcing both geometric separability and feature discrimination is essential for accurate fault localization.

V. CONCLUSION

We presented DSLA, a unified framework that bridges dynamic coverage and static dependencies through line-level graph reconstruction and a multi-objective Graph Attention Network. Extensive experiments demonstrate that DSLA is both robust and effective, surpassing state-of-the-art methods by over 1.3× on Defects4J and outperforming the baseline by an average of 6.1× across the top-10 most frequent defect types in Codeflaws. Further ablation studies show that the tight integration of execution traces, attention mechanisms, and geometric separability constraints is essential for achieving precise fault discrimination.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported in part by the Key Project of the National Natural Science Foundation of China (NSFC) under Grant Nos. 92467201 and 62402225.

REFERENCES

[1] S. Wang, J. Nam, and L. Tan, "Qtep: quality-aware test case prioritization," in *ESEC/FSE*. ACM, 2017, pp. 523–534.

[2] N. A. A. Khleel and K. Nehéz, "Software defect prediction using a bidirectional lstm network combined with oversampling techniques," *Cluster Computing*, vol. 27, no. 3, pp. 3615–3638, Oct. 2023.

[3] S. Qiu, H. Huang, W. Jiang, F. Zhang, and W. Zhou, "Defect prediction via tree-based encoding with hybrid granularity for software sustainability," *IEEE Transactions on Sustainable Computing*, vol. 9, no. 3, pp. 249–260, 2024.

[4] M. Zhou, H. Wang, K. Li, H. Zhu, and L. Sun, "Save the bruised striver: A reliable live patching framework for protecting real-world plcs," in *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, Athens, Greece, April 2024.

[5] M. Zhou, X. Hu, Z. Wang, H. Wang, H. Wen, L. Sun, and P. Zhang, "Dynamic vulnerability patching for heterogeneous embedded systems using stack frame reconstruction," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2025, p. 3855–3869.

TABLE III: Recall at Top-N comparison of DSLA and LLMAO on the Codeflaws dataset, categorized by fault type.

Fault type	Bug count	DSLA			LLMAO		
		TOP-1	TOP-3	TOP-5	TOP-1	TOP-3	TOP-5
DRVA	118	58.5%	87.3%	93.2%	4.2%	30.5%	41.5%
DCCR	249	53.0%	84.7%	89.6%	8.8%	29.3%	52.2%
DCCA	42	52.4%	90.5%	100.0%	7.1%	28.6%	40.5%
ORRN	187	62.0%	90.4%	93.0%	2.7%	16.6%	32.1%
OILN	162	61.1%	86.4%	92.0%	1.9%	17.9%	39.5%
OAIS	100	49.0%	84.0%	89.0%	11.0%	33.0%	55.0%
HIMS	101	56.4%	83.2%	94.1%	6.9%	26.7%	47.5%
HDMS	113	48.7%	82.3%	89.4%	22.1%	46.9%	64.6%
HOTH	163	61.3%	85.3%	93.9%	7.4%	29.4%	55.2%
STYP	58	56.9%	84.5%	91.4%	6.9%	22.4%	39.7%

TABLE IV: Ablation study on different encoders, feature types, and GNN types of DSLA. All models were trained with the combined 'MSE + t-SNE + Discriminability' loss function.

Static encoder	Coverage feature	GNN type	TOP-1	TOP-3	TOP-5	MFR	MAR
roberta	Yes	GATConv	58.2%	64.6%	66.1%	8.73	13.21
roberta	No	GATConv	24.6%	48.4%	53.4%	22.40	33.59
roberta	Yes	GCNConv	10.9%	23.3%	30.4%	26.76	35.53
codet5-large	Yes	GATConv	7.3%	21.0%	29.6%	24.54	33.42
codegen-350m	Yes	GATConv	7.9%	21.5%	29.4%	26.90	35.64

TABLE V: Ablation study of DSLA's loss function components. The base configuration is fixed: static encoder = `roberta`, coverage feature = `Yes`, and GNN type = `GATConv`.

Loss function	TOP-1	TOP-3	TOP-5	MFR	MAR
MSE + t-SNE + Discrim.	58.2%	64.6%	66.1%	8.73	13.21
MSE + t-SNE	34.1%	45.3%	49.7%	19.29	25.32
MSE + Discrim.	26.3%	37.8%	41.5%	24.26	30.08
MSE	15.9%	24.7%	28.8%	31.27	36.77

[6] X. Yu, J. Rao, L. Liu, G. Lin, W. Hu, J. W. Keung, J. Zhou, and J. Xiang, "Improving effort-aware defect prediction by directly learning to rank software modules," *Information and Software Technology*, vol. 165, p. 107250, 2024.

[7] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *ICSE*. ACM, 2016, pp. 1039–1050.

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[9] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *ESEC/FSE*. ACM, 2021, pp. 664–676.

[10] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *ICSE*. ACM, 2022, pp. 1169–1180.

[11] A. Z. H. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *ICSE*. ACM, 2024.

[12] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *ISSTA*. ACM, 2016, pp. 165–176.

[13] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[14] A. Beszédes, F. Horváth, M. Di Penta, and T. Gyimóthy, "Leveraging contextual information from function call chains to improve fault localization," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 468–479.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, 2009, pp. 103–116.

[16] X. Cheng, X. Nie, N. Li, H. Wang, Z. Zheng, and Y. Sui, "How about bug-triggering paths? - understanding and characterizing learning-based vulnerability detectors," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 2, pp. 542–558, 2024.

[17] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.

[18] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *PRDC*, 2006, pp. 39–46.

[19] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012, pp. 21–30.

[20] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*. New York, NY, USA: ACM, 2002, p. 467–477.

[21] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *ISSTA*. ACM, 2019, pp. 169–180.

[22] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *ICSE*, 2021, pp. 661–673.

[23] Y. Li, B. Cai, H. Yang, C. Cai, Y. Qiu, T. Chen, and C. Zhang, "Enhancing fault localization via causal measurement features," in *2025 International Joint Conference on Neural Networks (IJCNN)*, 2025, pp. 1–8.

[24] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (pca)," *Computers & Geosciences*, vol. 19, no. 3, pp. 303–342, 1993.

[25] M. Wattenberg, F. Viégas, and I. Johnson, "How to use t-sne effectively," *Distill*, 2016.

[26] GCC Developers, "Gcov," accessed: 2025-2-12. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[27] Cobertura Developers, "Cobertura," accessed: 2025-03-21. [Online]. Available: https://cobertura.github.io/cobertura/

[28] Joern Team, "Joern: A static analysis tool for codebases," https://github.com/joernio/joern, 2025.

[29] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019. [Online]. Available: https://arxiv.org/abs/1907.11692

[30] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *ISSTA*. ACM, 2014, pp. 437–440.

[31] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[32] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: an extensive study on 16 program repair systems," in *ASE*. ACM, 2021, pp. 907–918.