DCVD: Dual-Channel Cross-Modal Fusion for Joint Vulnerability Detection and Localization

Wenxin Tang[1,2,*] **Junliang Liu**[3,*] **Wenbin Li**[4] **Jingyu Xiao**[5] **Xi Xiao**[1,2,†]
Mingzhe Liu[6] **Jinlong Yang**[7] **Xuan Liu**[8] **Yuehe Ma**[9]
Wang Luo[10] **Qing Li**[2,†] **Lei Wang**[11] **Peng Xiangli**[12]

[1]Shenzhen International Graduate School, Tsinghua University
[2]Peng Cheng Laboratory [3]Dalian Maritime University
[4]Hunan University [5]The Chinese University of Hong Kong
[6]Shenzhen University [7]Northwestern Polytechnical University
[8]Shandong University [9]BNU-HKBU United International College
[10]Sun Yat-sen University [11]Guangzhou Intelligence Communications Technology Co., Ltd.
[12]The Fifth Electronic Research Institute of MIIT

twx24@mails.tsinghua.edu.cn 11202412921jl@dlmu.edu.cn xiaox@sz.tsinghua.edu.cn

Abstract

Software vulnerability detection plays a critical role in ensuring system security, where real-world auditing requires not only determining whether a function is vulnerable but also pinpointing the specific lines responsible. However, existing approaches either rely on a single information source—sequential, structural, or semantic—failing to jointly exploit the complementary strengths across modalities, or treat statement-level localization merely as a byproduct of function-level detection without explicit line-level supervision. To address these limitations, we propose DCVD (Dual-Channel Cross-Modal Vulnerability Detection), a unified framework that performs joint function-level detection and statement-level localization. DCVD extracts control-dependency and semantic features through two parallel branches and integrates them via contrastive alignment coupled with bidirectional cross-attention, effectively bridging the cross-modal representation gap. It further introduces explicit supervision signals at both the function and statement levels, enabling collaborative optimization across the two granularities. Extensive experiments on three real-world vulnerability benchmarks demonstrate that DCVD consistently outperforms state-of-the-art methods on all three benchmarks. Our code is available at `https://github.com/vinsontang1/DCVD`.

1 Introduction

The increasing scale and complexity of modern software systems have made security vulnerabilities a pervasive and critical concern. From buffer overflows (Cowan et al., 2000; Lhee and Chapin, 2003) and use-after-free (Caballero et al., 2012; Lee et al., 2015) errors to injection attacks (Zou et al., 2025; Ray and Ligatti, 2012) and authentication bypasses (Shoshitaishvili et al., 2015), software vulnerabilities (Sánchez et al., 2019) pose severe threats to system integrity, user privacy, and economic security. As software development accelerates driven by agile methodologies and open-source ecosystems, the volume of code requiring security review has far outpaced the capacity of manual auditing. Consequently, automated vulnerability detection has emerged as an indispensable capability in the software security lifecycle. Recent advances in deep learning and large language models (LLMs) have demonstrated promising results in software understanding tasks (Liu et al.,

*Equal contribution.
†Corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

2025; Xiao et al., 2026c), opening new avenues for learning-based vulnerability detection. However, real-world security auditing demands not only determining *whether* a function contains a vulnerability (function-level detection), but also pinpointing *which specific lines* are responsible (statement-level localization). Satisfying both requirements simultaneously remains an open challenge.

Existing approaches to vulnerability detection can be broadly categorized into three paradigms based on their primary information source. **(a) Sequence-based methods** model source code as token sequences and leverage Transformer architectures to capture contextual semantics (Liu et al., 2024; Wen et al., 2023; Du et al., 2024). For instance, LineVul (Fu and Tantithamthavorn, 2022) applies a pre-trained RoBERTa model to learn token-level representations for vulnerability prediction. While such methods effectively capture sequential semantic patterns, they inherently discard the structural information embedded in program control flow and syntax hierarchies. **(b) Graph-based methods** take program representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Program Dependence Graphs (PDGs) as input to capture structural dependencies within code (Wang et al., 2024; Zhang et al., 2023; Weng et al., 2024; Peng et al., 2023). Representative works including MatsVD (Weng et al., 2024) and PTLVD (Peng et al., 2023) have shown that structural information provides valuable signals for vulnerability detection. However, these methods primarily focus on syntactic and control-flow patterns while lacking the ability to comprehend high-level semantic intent and functional logic of the code. **(c) LLM-based methods** directly employ Code LLMs for vulnerability detection, leveraging their strong semantic understanding capabilities acquired through large-scale pre-training on code corpora (Wen et al., 2024; Yang et al., 2024; Du et al., 2024), and recent frameworks further orchestrate multiple LLM agents to reason about vulnerabilities (Widyasari et al., 2026; Tsai et al., 2025). Despite their powerful language comprehension, these approaches lack explicit modeling of program structural properties, limiting their sensitivity to vulnerabilities rooted in control flow anomalies. These three paradigms each rely on a single information source and fail to effectively integrate structural and semantic information, resulting in limited performance when confronting complex vulnerability patterns that simultaneously involve control flow irregularities and semantic anomalies. Moreover, existing frameworks that attempt both function-level detection and statement-level localization, such as PTLVD (Peng et al., 2023) and LineVul (Fu and Tantithamthavorn, 2022), treat statement-level localization merely as a byproduct of function-level detection—typically via attention weights or gradient-based attribution—without explicit statement-level supervision, leading to insufficient localization precision.

To advance automated vulnerability detection, three key technical challenges must be addressed: **(1) How to effectively extract vulnerability-relevant control dependency information.** Vulnerabilities in real-world software often do not originate from isolated statements but arise from complex interactions among control dependencies—a missing boundary check may render a subsequent array access unsafe, or an incorrectly configured conditional branch may expose an unauthorized execution path. These vulnerabilities are deeply embedded within control flow structures, making them difficult to identify through surface-level analysis. Existing static analysis techniques provide coarse-grained representations of control dependencies that lack the flexibility to distinguish, within complex nested structures and conditional jumps, which control paths are critical for vulnerability triggering and which constitute irrelevant noise. **(2) How to bridge the cross-modal representation gap between structural and semantic features.** Control dependency information is naturally represented as graph structures emphasizing syntactic and flow relationships, while semantic information manifests as continuous embedding vectors capturing functional intent and behavioral logic. These two modalities exhibit fundamentally different data formats, semantic granularities, and feature distributions. Naive fusion strategies such as concatenation or weighted averaging fail to achieve meaningful alignment and may introduce representational conflicts that degrade detection performance. **(3) How to reconcile the supervision asymmetry between function-level detection and statement-level localization.** Function-level detection constitutes a coarse-grained binary classification task, whereas statement-level localization requires fine-grained sequence-level prediction. The two tasks differ substantially in label density and gradient signal strength. Existing methods lack explicit statement-level supervision, relegating localization to an auxiliary output derived indirectly from function-level predictions, which fundamentally constrains localization accuracy.

To address these challenges, we propose **DCVD** (**D**ual-**C**hannel Cross-Modal **V**ulnerability **D**etection), a unified framework that synergistically integrates multi-source information extraction, cross-modal feature fusion, and multi-granularity supervised learning for joint vulnerability detection and localization. To tackle **Challenge (1)**, we propose the *Dual-Channel Encoder*, which comprises

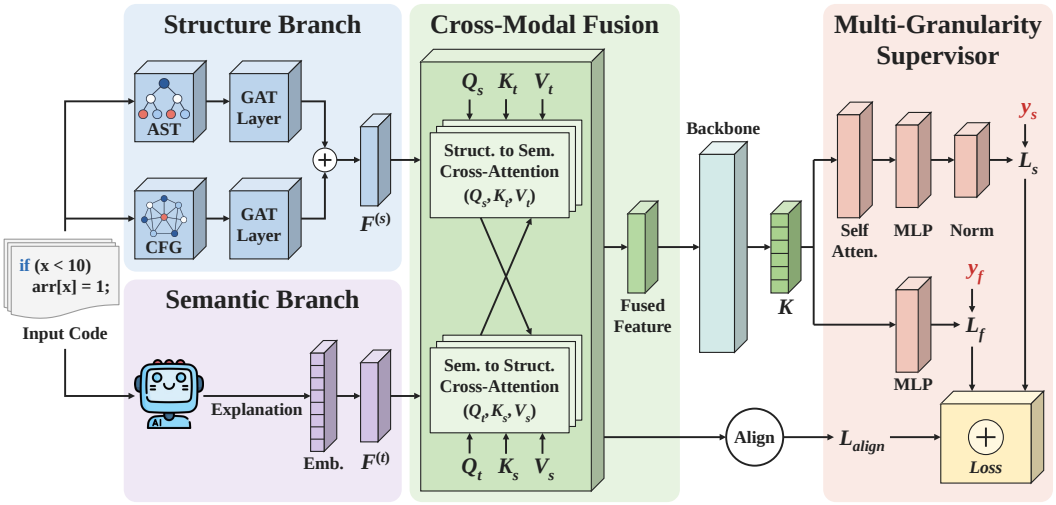


Figure 1: The framework of DCVD.

a structure branch and a semantic branch operating in parallel: the former employs Graph Attention Networks (GATs) to extract control dependency features from AST and CFG representations, automatically focusing on vulnerability-relevant control nodes and paths; the latter leverages an LLM to generate natural language explanations of the code and extracts deep semantic features through a shared embedding layer. The two branches provide complementary coverage of both structural and semantic dimensions. To address **Challenge (2)**, we introduce the *Cross-Modal Fusion* module, which first aligns the two modalities into a unified high-dimensional space via contrastive learning, and then employs bidirectional cross-attention to enable deep interaction and mutual enhancement, producing a fused representation that integrates both structural and semantic information. To resolve **Challenge (3)**, we design the *Multi-Granularity Supervisor*, which comprises function-level and statement-level supervision branches operating in parallel: the former optimizes overall vulnerability detection through classification loss, while the latter incorporates explicit statement-level supervision signals for fine-grained localization. The two branches are jointly optimized to achieve collaborative detection and localization. Additionally, we employ a Transformer-based pre-trained language model to perform deep contextual modeling on the fused features before the supervision module, further enhancing its capacity to capture long-range dependencies and cross-line context.

Our main contributions are summarized as follows:

- We propose the *Dual-Channel Encoder* that combines GAT-based control dependency feature extraction with LLM-enhanced semantic feature extraction in a parallel architecture, effectively capturing multi-dimensional information closely related to vulnerabilities from both structural and semantic perspectives.
- We introduce the *Cross-Modal Fusion* module based on contrastive learning and bidirectional cross-attention, which bridges the representation gap between graph-structural and textual-semantic features and achieves deep complementary fusion of the two information sources.
- We design the *Multi-Granularity Supervisor* that incorporates explicit statement-level supervision alongside function-level supervision, enabling collaborative optimization of vulnerability detection and localization without relying on indirect attention-based attribution.
- Experiments show that DCVD outperforms state-of-the-art methods on all three benchmarks.

2 Related Work

2.1 Semantic-Based Vulnerability Detection

With the strong understanding capabilities demonstrated by large language models (Yu et al., 2026; Tang et al., 2025, 2026; Luo et al., 2026), semantic modeling based on pre-trained models and LLMs

has been widely applied to code-related tasks (Xiao et al., 2025a, 2026a), including vulnerability detection. Early work treats source code as token sequences and leverages pre-trained Transformers for modeling. LineVul (Fu and Tantithamthavorn, 2022) adopts CodeBERT (Feng et al., 2020) with BPE tokenization (Berglund and van der Merwe, 2023) for function-level vulnerability prediction, and infers line-level results through self-attention weights; VulBERTa (Hanif and Maffeis, 2022) performs lightweight RoBERTa (Liu et al., 2019) pre-training under a customized tokenization scheme that combines BPE with predefined tokens, achieving strong performance under limited data budgets. Along the direction of semantic enrichment, SCALE (Wen et al., 2024) attaches natural language comments generated by LLMs onto ASTs to compensate for the limited understanding of complex statements in pre-trained models; Counterfactual (Kuang et al., 2023) randomizes the replacement of user-defined identifiers to generate counterfactual data that mitigates superficial spurious correlations; More recent work further exploits the reasoning capabilities of LLMs: VulLLM (Du et al., 2024) employs multi-task instruction tuning on detection, localization, and explanation to force LLMs to learn the root causes of vulnerabilities; MSIVD (Yang et al., 2024) jointly trains an LLM fine-tuned with multi-turn self-instructed dialogues together with a data-flow GNN adapter. Agent-based frameworks prompt LLMs without fine-tuning: VulTrial (Widyasari et al., 2026) stages a mock-court debate among role-specific agents, and ConColl (Tsai et al., 2025) escalates from a single agent to retrieval-augmented multi-agent reasoning; both output function-level verdicts only. Nevertheless, these methods primarily rely on semantics at the token or natural language level, and remain limited in modeling the control and data dependencies inherent to code.

2.2 Structure-Aware Vulnerability Detection

Another line of research explicitly injects code structure or execution dependencies into models to capture vulnerability patterns that are difficult to characterize through semantics alone. Classical static analysis merges ASTs, CFGs, and PDGs into Code Property Graphs (CPGs) and matches hand-written graph queries (Yamaguchi et al., 2014; Joern Contributors, 2026), which is efficient but covers only predefined patterns. Along the direction of graph and dependency modeling, DFEPT (Jiang et al., 2024) extracts data-flow graphs from ASTs and augments pre-trained models with GNNs as a plug-and-play component; PDBERT (Liu et al., 2024) introduces control-dependency and data-dependency prediction as new pre-training objectives; MatsVD (Weng et al., 2024) converts the dependency relations in PDGs into attention masks that constrain Transformers to attend only among dependent statements. For fine-grained output, PTLVD (Peng et al., 2023) combines program slicing with Transformers and aggregates token-level contributions from Integrated Gradients and Saliency into line-level scores. Despite the progress in either structural modeling or fine-grained output, most of these methods are confined to a single information source, or treat statement-level localization as a post-hoc explanation of function-level predictions, lacking deep integration of structural and semantic information as well as explicit line-level supervision.

3 Methodology

3.1 Overview

To jointly achieve function-level vulnerability detection and statement-level localization, we propose the DCVD framework, as illustrated in Figure 1. Given an input function, the Dual-Channel Encoder first extracts code features from both structural and semantic perspectives in parallel, producing $F^{(s)}$ and $F^{(t)}$ respectively. The Cross-Modal Fusion module then aligns the two representations via contrastive learning and fuses them through bidirectional cross-attention to obtain the fused feature H. A multi-layer Transformer subsequently performs deep contextual modeling on H, capturing long-range dependencies and cross-line contextual information to yield the enriched representation K. Finally, the Multi-Granularity Supervisor conducts joint supervised training on K for both function-level detection and statement-level localization.

3.2 Dual-Channel Encoder

Structure Branch. The structure branch captures control dependency information critical for identifying vulnerabilities rooted in control flow anomalies. We employ static analysis tools to extract two complementary graph representations from the source code: the AST encoding hierarchical

syntactic structure, and the CFG capturing execution flow and branching logic. Both share the same node set but differ in edge connectivity.

Given node features $X \in \mathbb{R}^{N \times d}$ obtained through a learnable embedding layer, we process the AST and CFG through two independent GAT (Veličković et al., 2017) networks. For each GAT, the attention coefficient between node i and its neighbor $j \in \mathcal{N}(i)$ is:

$$\alpha_{ij} = \frac{\exp\Big(\rho\big(a^\top [Wx_i \| Wx_j]\big)\Big)}{\sum_{k \in \mathcal{N}(i)} \exp\Big(\rho\big(a^\top [Wx_i \| Wx_k]\big)\Big)} \tag{1}$$

where W is a learnable weight matrix, a is the attention parameter vector, $\|$ denotes concatenation, and ρ is a nonlinear activation function. The updated representation of node i is obtained by aggregating neighbor features weighted by the attention coefficients:

$$x_i' = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}\, W x_j\right) \tag{2}$$

where σ denotes an activation function. Through this mechanism, the GAT adaptively assigns higher weights to control nodes most relevant to vulnerability patterns, such as boundary checks and conditional branches, while suppressing noise from irrelevant structural elements.

Let f_A and f_C denote the two GAT networks operating on the AST and CFG respectively. The structural representation is obtained by element-wise addition:

$$F^{(s)} = f_A(X, E_A) + f_C(X, E_C) \tag{3}$$

where E_A and E_C are the edge index matrices. The resulting $F^{(s)} \in \mathbb{R}^{N \times d'}$ encodes both syntactic hierarchies and control flow dependencies.

Semantic Branch. The semantic branch extracts features capturing functional intent, behavioral logic, and API call semantics that are difficult to obtain from structural analysis alone. We first feed the preprocessed source code into an LLM to generate a natural language explanation describing the code's functionality and potential security-relevant behaviors. This explanation surfaces implicit logic and external API semantics not directly expressed in the code syntax.

Both the source code and its explanation are tokenized and passed through a shared embedding layer $\mathcal{E}$ extracted from a pre-trained language model. Let $C = \mathcal{E}(c) \in \mathbb{R}^{M \times d_h}$ and $T = \mathcal{E}(t) \in \mathbb{R}^{L \times d_h}$ denote the code and explanation embeddings respectively, where M and L are the sequence lengths. We compute a global explanation vector via masked mean pooling:

$$\bar{t} = \frac{\sum_{j=1}^{L} T_j \cdot m_j}{\sum_{j=1}^{L} m_j} \tag{4}$$

where m_j is the attention mask indicating valid tokens. The semantic representation is obtained by injecting the global explanation into each code token and projecting to the target dimension: $F^{(t)} = \phi(C + \bar{t})$, where ϕ is a nonlinear projection mapping to dimension d'. The resulting $F^{(t)} \in \mathbb{R}^{M \times d'}$ incorporates both token-level code semantics and the global functional understanding provided by the LLM explanation.

3.3 Cross-Modal Fusion

The Dual-Channel Encoder produces two complementary yet heterogeneous representations: $F^{(s)}$ from graph neural networks and $F^{(t)}$ from language model embeddings. These modalities reside in different representation subspaces, and direct concatenation fails to exploit their complementary nature. We adopt a two-stage strategy: contrastive alignment followed by bidirectional cross-attention.

Contrastive Alignment. We first align the two modalities into a shared space through contrastive learning. For each sample i in a mini-batch of size B, we derive global representations g_i from $F^{(s)}$ via mean pooling over graph nodes, and c_i from the semantic branch. The pair (g_i, c_i) from the same function forms a positive pair, while cross-function pairs serve as negatives. The alignment loss is

$$\mathcal{L}_{align} = -\frac{1}{B} \sum_{i=1}^{B} \log \frac{\exp(\mathrm{sim}(g_i, c_i)/\tau)}{\sum_{j=1}^{B} \exp(\mathrm{sim}(g_i, c_j)/\tau)} \tag{5}$$

where $\mathrm{sim}(\cdot,\cdot)$ denotes cosine similarity and τ is a temperature hyperparameter. This objective pulls same-function representations together while pushing different-function representations apart, reducing the cross-modal distribution gap before fusion.

Bidirectional Cross-Attention. After alignment, we employ bidirectional cross-attention to enable deep interaction between the two modalities. Unlike self-attention, cross-attention derives queries from one modality and keys/values from the other, allowing each modality to selectively attend to the most informative elements of its counterpart.

In the structure-to-semantic direction, the structural representation queries the semantic feature: $H_s = \mathrm{softmax}\Big(Q_s {K_t}^\top / \sqrt{d'}\Big)\, V_t$, where $Q_s = F^{(s)} W_Q^s$, $K_t = F^{(t)} W_K^t$, and $V_t = F^{(t)} W_V^t$ are linear projections. In the semantic-to-structure direction: $H_t = \mathrm{softmax}\Big(Q_t {K_s}^\top / \sqrt{d'}\Big)\, V_s$, where $Q_t = F^{(t)} W_Q^t$, $K_s = F^{(s)} W_K^s$, and $V_s = F^{(s)} W_V^s$. Since $H_s \in \mathbb{R}^{N \times d'}$ operates at the graph-node level while $H_t \in \mathbb{R}^{M \times d'}$ operates at the token level, we apply mean pooling to H_s and broadcast the resulting vector to match the token sequence length. The fused feature is then:

$$H = \sigma\big([H_t \| \bar{H}_s] W_m^\top\big) \tag{6}$$

where $W_m \in \mathbb{R}^{d' \times 2d'}$ is a learnable projection, $\bar{H}_s$ denotes the broadcast vector, and σ is an activation function. The resulting $H \in \mathbb{R}^{M \times d'}$ enables each token position to be informed by both control dependency context and semantic intent through mutual querying.

After that, we employ a multi-layer Transformer to perform deep contextual modeling on H, capturing long-range dependencies and cross-line contextual patterns. Specifically, the fused embeddings are projected to match the hidden dimension and fed into the Transformer encoder, yielding $K = f_{\mathrm{T}}(W_a\, H)$, where W_a is a dimension alignment projection and f_{T} denotes the multi-layer Transformer. The output $K \in \mathbb{R}^{M \times d_k}$ is fed to both branches of the Multi-Granularity Supervisor.

3.4 Multi-Granularity Supervisor

To jointly optimize vulnerability detection and localization under a unified framework, we design the Multi-Granularity Supervisor, which comprises two parallel branches: a function-level detection branch and a statement-level localization branch, each equipped with its own supervision signal.

Function-Level Detection. The function-level branch leverages function-level labels to determine whether the input function contains a vulnerability. The sequence representation K is pooled into a global vector and passed through an MLP g_f to produce a binary prediction:

$$\hat{y}_f = \sigma\big(g_f\big(\bar{K}\big)\big) \tag{7}$$

where $\bar{K}$ denotes the pooled representation and σ is the sigmoid function. The function-level loss $\mathcal{L}_f$ is computed using binary cross-entropy between $\hat{y}_f$ and the ground-truth label y_f.

Statement-Level Localization. The statement-level branch leverages line-level labels to identify the specific lines responsible for the vulnerability. We first refine the token-level representations through a self-attention layer f_{sa} followed by an MLP g_s and layer normalization:

$$\tilde{K} = \mathrm{LN}(K + g_s(f_{sa}(K))) \tag{8}$$

Each token representation is then projected to a scalar vulnerability score. Since labels are annotated at the line level, we aggregate token scores within each line by averaging:

$$s_l = \frac{1}{|\mathcal{T}_l|} \sum_{i \in \mathcal{T}_l} w^\top \tilde{K}_i \tag{9}$$

where $\mathcal{T}_l$ is the set of token indices belonging to line l and w is a learnable projection vector. The line-level vulnerability probability is $\hat{y}_s^{(l)} = \sigma(s_l)$. The statement-level loss $\mathcal{L}_s$ is the binary cross-entropy between $\hat{y}_s^{(l)}$ and the ground-truth line label $y_s^{(l)} \in \{0, 1\}$, averaged over all lines of the function.

We combine both branch losses with the contrastive alignment loss to form the total training objective:

$$\mathcal{L} = \alpha\,(\mathcal{L}_f + \beta \cdot \mathcal{L}_{align}) + (1 - \alpha) \cdot \mathcal{L}_s \tag{10}$$

where α and β are balancing hyperparameters. The function-level branch provides global guidance on vulnerable regions, the statement-level branch delivers fine-grained localization through explicit supervision, and the alignment loss ensures cross-modal consistency throughout training.

4 Experiments

4.1 Experimental Setup

Dataset. We conduct experiments on BigVul (Fan et al., 2020), a large-scale real-world C/C++ vulnerability dataset that provides both function-level vulnerability labels and statement-level annotations indicating the specific lines responsible for each vulnerability. We also evaluate function-level detection on PrimeVul (Ding et al., 2025) and VulGate (Safdar et al., 2026). Following LineVul (Fu and Tantithamthavorn, 2022), we split BigVul into training, validation, and test sets at a ratio of 8:1:1, and we adopt the official 8:1:1 splits of PrimeVul and VulGate, evaluating PrimeVul on its paired test set. DCVD and the learning-based baselines are trained on the training sets and evaluated on the test sets.

Baselines. We compare DCVD against twelve representative vulnerability detection methods. (1) **TextCNN** (Guo et al., 2019) applies multi-kernel convolutions over token sequences for sequence-level classification. (2) **LineVul** (Fu and Tantithamthavorn, 2022) fine-tunes a Transformer-based pre-trained model on source code tokens and derives statement-level predictions from attention weights. (3) **IVDetect** (Li et al., 2021) employs a feature-attention GCN over program dependence graphs enriched with multi-type contextual features. (4) **LineVD** (Hin et al., 2022) formulates statement-level localization as a node classification task on statement-level graphs. (5) **VELVET** (Ding et al., 2022) ensembles a Transformer and a gated graph neural network to jointly exploit sequential and structural information. (6) **ICVH** (Nguyen et al., 2021) combines code representation learning with a hierarchical classification architecture for vulnerability identification. (7) **MatsVD** (Weng et al., 2024) adopts a multi-task architecture that jointly performs function-level detection and statement-level localization. (8) **Static Analysis** follows the code property graph paradigm (Yamaguchi et al., 2014): Joern (Joern Contributors, 2026) builds a CPG that unifies AST, CFG, and data dependencies, and graph-query rules cover common categories such as buffer and integer overflows. (9) **VulTrial** (Widyasari et al., 2026) and (10) **ConColl** (Tsai et al., 2025) are multi-agent LLM detectors that output function-level predictions only. (11) **GPT-5.5** (OpenAI, 2026) directly prompts a frontier LLM, and (12) **Codex + GPT-5.5** (OpenAI, 2025) applies an off-the-shelf coding agent. VulTrial and ConColl use their original prompts, while GPT-5.5 and Codex + GPT-5.5 are instructed to reason step by step without network retrieval (Appendix H.2). Due to API cost and following common practice, these four LLM-based methods are evaluated on the complete paired test set of PrimeVul and on the same 1,000 functions randomly sampled from the BigVul and VulGate test sets, while all other methods use the complete test sets.

Evaluation Metrics. We evaluate DCVD from two complementary perspectives. For *classification quality*, we report Matthews Correlation Coefficient (Mcc), Precision (Pre), Recall (Re), and F1-score at both the function level and the statement level. Statement-level classification is further evaluated under two protocols: the *Two-Phase* setting, where statement-level prediction is conducted only on functions predicted as vulnerable by the function-level branch, and the *One-Phase* setting, where statement-level prediction is conducted independently of the function-level branch. For *localization quality*, we adopt Top-1, Top-3, and Top-5 Accuracy, Mean First Rank (MFR), and Mean Average Rank (MAR), which jointly characterize how precisely a method ranks vulnerable lines among all lines in a function. On PrimeVul we follow its protocol and report Pre, Re, False Positive Rate (FPR), F1, and Accuracy (Acc); on VulGate we report Pre, Re, and F1.

The semantic branch of DCVD uses GPT-4o-mini-2024-07-18 to produce natural language explanations of the input code, and the multi-layer Transformer for deep contextual modeling is instantiated as GraphCodeBERT (Guo et al., 2020). DCVD is trained end-to-end on NVIDIA H20 GPUs.

4.2 Main Results

Table 1 presents the classification performance at both function and statement levels, and Table 2 presents the statement-level localization ranking performance on BigVul; Table 3 presents the function-level results on PrimeVul and VulGate, evaluated using the metrics introduced in Section 4.1. Due to computational cost, we evaluate all baselines on BigVul, while on PrimeVul and VulGate we select the best-performing learning-based baseline, MatsVD.

DCVD outperforms the baselines on nearly all metrics. For function-level detection, the largest gains over the strongest baseline MatsVD are observed on F1 and Recall, which improve from 93.82 and

Table 1: Function-level and statement-level classification performance on BigVul. Statement-level results are reported under both Two-Phase and One-Phase protocols.

Method	Two-Phase Detection								One-Phase Detection			
	Function Level				Statement Level				Statement Level			
	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1
TextCNN	0.3856	61.76	26.23	36.82	0.1404	21.10	30.15	24.83	0.2205	26.24	40.27	31.78
ICVH	0.4556	65.73	33.94	44.76	0.1804	21.39	43.85	28.75	0.2656	27.34	50.76	35.54
IVDetect	0.6171	72.28	55.23	62.62	0.3014	28.36	58.46	38.19	0.3564	31.52	65.06	42.47
LineVD	0.7537	82.47	70.76	76.17	0.3899	35.52	63.86	45.65	0.4506	40.94	67.38	50.93
LineVul	0.8224	87.65	78.58	82.87	0.4951	62.10	47.45	53.79	0.5299	62.38	53.40	57.54
VELVET	0.8973	95.42	85.20	90.02	0.6622	68.74	71.23	69.96	0.6781	59.87	86.25	70.68
MatsVD	0.9361	**97.53**	90.37	93.82	0.8463	91.57	81.13	86.03	0.8585	89.90	84.86	87.31
Static Analysis	0.3311	42.35	29.96	35.10	0.1646	22.90	32.49	26.87	0.1239	17.99	37.85	24.39
GPT-5.5	0.7737	87.23	70.69	78.10	0.7146	77.30	71.71	74.40	0.7159	73.62	75.88	74.73
ConColl	0.8719	87.93	87.93	87.93	–	–	–	–	–	–	–	–
Codex + GPT-5.5	0.8784	90.91	86.21	88.50	0.8303	88.15	81.58	84.74	0.8436	84.71	**87.50**	86.08
VulTrial	0.9250	96.30	89.66	92.86	–	–	–	–	–	–	–	–
DCVD (ours)	**0.9461**	97.46	**92.30**	**94.81**	**0.8694**	**92.59**	**84.18**	**88.18**	**0.8734**	**92.70**	84.78	**88.56**

Table 2: The statement-level localization ranking performance on BigVul.

Method	Top-1	Top-3	Top-5	MFR	MAR
TextCNN	27.44	34.78	45.97	13.16	18.69
ICVH	35.38	43.08	55.84	11.80	16.24
IVDetect	46.69	54.27	62.94	8.73	12.63
LineVD	60.53	73.77	80.75	4.63	10.21
LineVul	24.55	48.50	60.65	9.32	13.94
VELVET	86.04	92.42	93.86	3.73	7.81
MatsVD	92.78	94.46	94.83	2.46	6.34
Static Analysis	9.75	17.69	25.39	17.68	23.76
GPT-5.5	68.97	87.93	96.55	4.28	7.95
Codex + GPT-5.5	87.93	96.55	96.55	2.86	6.00
DCVD (ours)	**93.74**	**96.99**	**97.83**	**1.77**	**4.17**

90.37 to 94.81 and 92.30 respectively (+0.99 and +1.93), confirming that integrating complementary structural and semantic information through the Dual-Channel Encoder and Cross-Modal Fusion effectively overcomes the limitations of methods relying on a single information source.

Benefiting from the explicit statement-level supervision introduced by the Multi-Granularity Supervisor, DCVD also leads on statement-level classification under both the Two-Phase and One-Phase protocols. Under Two-Phase, Mcc and F1 improve from 0.8463 and 86.03 to 0.8694 and 88.18 (+0.0231 and +2.15); under One-Phase, they improve from 0.8585 and 87.31 to 0.8734 and 88.56 (+0.0149 and +1.25). The gains under both protocols indicate that the statement-level branch of DCVD is not merely a byproduct of function-level prediction but is independently discriminative.

On the ranking metrics, DCVD achieves the best performance across Top-1/3/5, MFR, and MAR. In particular, relative to MatsVD, Top-1 increases from 92.78 to 93.74 (+0.96), MFR decreases from 2.46 to 1.77 (-0.69 positions), and MAR decreases from 6.34 to 4.17 (-2.17 positions). This demonstrates that DCVD ranks truly vulnerable lines substantially closer to the top, rather than relying on indirect attention- or gradient-based attribution as in prior work.

Overall, the results across classification and ranking metrics show that DCVD consistently improves both function-level detection and statement-level localization over existing methods on BigVul, and DCVD also achieves the best F1 on PrimeVul and VulGate (Table 3).

Table 3: Function-level detection results on PrimeVul and VulGate.

Method	PrimeVul					VulGate		
	Pre	Re	FPR	F1	Acc	Pre	Re	F1
MatsVD	57.75	43.68	31.95	49.74	55.86	79.22	88.30	83.51
Static Analysis	51.06	16.55	**15.86**	25.00	50.34	47.46	23.56	31.49
GPT-5.5	58.07	51.26	37.01	54.46	57.13	79.19	69.83	74.22
ConColl	51.39	**84.83**	80.23	64.01	52.30	89.98	87.13	88.53
Codex + GPT-5.5	57.23	64.60	48.28	60.69	58.16	85.68	83.33	84.49
VulTrial	54.22	62.07	52.41	57.88	54.83	90.95	86.92	88.89
DCVD (ours)	**70.27**	71.72	30.34	**70.99**	**70.69**	**92.21**	**90.03**	**91.11**

Table 4: The function-level and statement-level classification performance of ablation study.

Setting	Two-Phase Detection								One-Phase Detection			
	Function Level				Statement Level				Statement Level			
	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1
Ours	**0.9379**	96.56	**91.67**	**94.05**	**0.8581**	**91.58**	**83.17**	**87.17**	**0.8649**	**91.82**	**84.14**	**87.81**
w/o Structure	0.8802	95.76	81.88	88.28	0.5852	74.15	52.22	61.28	0.6064	77.12	53.29	63.03
w/o Semantic	0.8679	94.49	80.80	87.11	0.5383	74.72	44.31	55.63	0.5701	78.99	46.20	58.30
w/o Fusion	0.8410	**96.70**	74.28	84.02	0.5020	83.04	34.22	48.47	0.5266	84.43	36.77	51.23
w/o Stmt-Level Task	0.8395	89.16	80.43	84.57	–	–	–	–	–	–	–	–
w/o Func-Level Task	–	–	–	–	–	–	–	–	0.7678	84.74	73.89	78.94

4.3 Ablation Study

To validate each component of DCVD, we design five ablation variants. **w/o Structure** removes the entire structure branch, relying solely on the semantic branch. **w/o Semantic** removes the semantic branch and retains only the structural branch. **w/o Fusion** replaces the contrastive alignment and bidirectional cross-attention with simple concatenation followed by an MLP. **w/o Statement-Level Task** removes the statement-level supervision, reducing the Multi-Granularity Supervisor to a single function-level branch (function-level only). Symmetrically, **w/o Function-Level Task** removes the function-level supervision; lacking function-level predictions, it supports only One-Phase and ranking evaluation. We randomly sample 30% of the test set for evaluation, with classification and ranking results reported in Table 4 and Table 5.

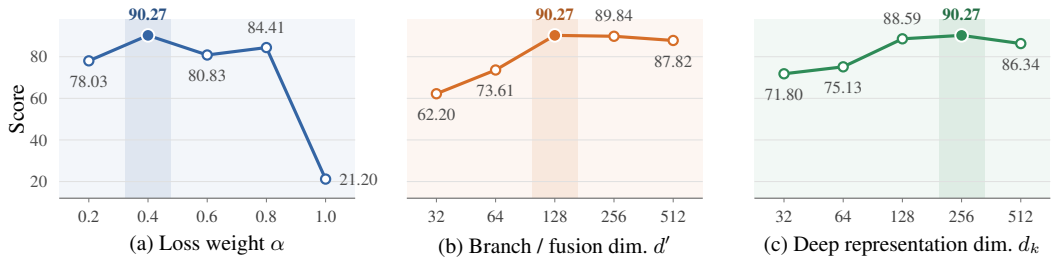
After removing each component, the performance exhibits varying degrees of decline across function-level and statement-level metrics, demonstrating the contribution of each module to the overall framework. In particular, **w/o Fusion** causes the largest drop among the feature-related variants on statement-level classification, confirming that naive concatenation is insufficient to bridge the heterogeneous subspaces of graph-structural and textual-semantic representations. More notably, after removing the statement-level supervision signal in **w/o Statement-Level Task**, the function-level metrics also drop substantially (F1: 94.05 $\rightarrow$ 84.57), indicating that statement-level supervision not only enables fine-grained localization but also provides auxiliary signals that reinforce function-level classification—thereby validating the synergistic design of the Multi-Granularity Supervisor. Conversely, **w/o Function-Level Task** lowers One-Phase statement-level F1 from 87.81 to 78.94 and Top-1 from 93.12 to 84.42, so the two granularities reinforce each other.

4.4 Parameter Analysis

We further investigate the impact of three key hyperparameters on DCVD, including α (the balancing coefficient between function-level and statement-level losses), d' (the output dimension of both branches and the fused representation), and d_k (the deep representation dimension produced by the multi-layer Transformer). For ease of visualization, we aggregate four representative metrics into a composite Score $=$ (Two-Phase Func F1 $+$ Two-Phase Stmt F1 $+$ One-Phase Stmt F1 $+$ Top-1)$/4$. The parameter analysis is conducted on the BigVul validation set, which is used to select the hyperparameters, and the results are shown in Figure 2; when one hyperparameter is varied, the other two are fixed to their default values in Table 6.

Table 5: The statement-level localization ranking performance of ablation study.

Setting	Top-1	Top-3	Top-5	MFR	MAR
Ours	**93.12**	**96.74**	**97.46**	**1.82**	**4.22**
w/o Structure	80.80	87.68	89.13	7.89	12.09
w/o Semantic	79.71	85.87	88.04	8.05	12.45
w/o Fusion	79.35	85.14	87.68	8.08	12.52
w/o Stmt-Level Task	–	–	–	–	–
w/o Func-Level Task	84.42	89.86	92.03	6.33	10.15



(a) Loss weight α (b) Branch / fusion dim. d' (c) Deep representation dim. d_k

Figure 2: Parameter sensitivity of DCVD: composite Score w.r.t. (a) the loss-balancing coefficient α, (b) the branch/fusion dimension d', and (c) the deep representation dimension d_k.

As illustrated in the figure, the Score peaks when $\alpha = 0.4$; a smaller α provides insufficient function-level supervision, while $\alpha = 1$ removes $\mathcal{L}_s$ entirely, so the model produces no statement-level predictions and the two statement-level F1 terms and Top-1 are counted as 0 in the Score. For d', the Score reaches its maximum of 90.27 at $d' = 128$: too small a dimension leaves the two branches and the fused representation without enough capacity to carry cross-modal information, while too large a dimension introduces redundant parameters and exacerbates overfitting. Regarding d_k, the best Score of 90.27 is obtained at $d_k = 256$, as a smaller d_k cannot adequately accommodate the long-range dependencies and cross-line context captured by the multi-layer Transformer, whereas a larger d_k comes with an increased risk of overfitting.

5 Conclusion

In this paper, we propose DCVD, a unified framework for joint function-level vulnerability detection and statement-level localization. DCVD extracts complementary structural and semantic representations through a Dual-Channel Encoder, bridges the cross-modal gap via contrastive alignment coupled with bidirectional cross-attention in the Cross-Modal Fusion module, and introduces a Multi-Granularity Supervisor that explicitly supervises both granularities to enable collaborative optimization. Two core insights underpin DCVD: deep cross-modal fusion between structural and semantic representations, and explicit supervision at both the function and statement levels, which together address the limitations of single-source modeling and attention-based line-level attribution. Comprehensive evaluations across classification, ranking, ablation, and parameter analyses consistently validate the effectiveness of each design choice. Extensive experiments on BigVul, PrimeVul, and VulGate demonstrate that DCVD consistently outperforms state-of-the-art methods on all three benchmarks.

Acknowledgments and Disclosure of Funding

This work is supported by the Natural Science Foundation of Guangdong Province (Grant No. 2025A1515011946) and the Basic and Frontier Research Project of PCL (Grant No. 2025QYA001).

The authors declare no competing interests.

References

Martin Berglund and Brink van der Merwe. Formalizing bpe tokenization. *arXiv preprint arXiv:2309.08715*, 2023.

William Brannon, Wonjune Kang, Suyash Fulay, Hang Jiang, Brandon Roy, Deb Roy, and Jad Kabbara. Congrat: Self-supervised contrastive pretraining for joint graph and text embeddings. In *Proceedings of TextGraphs-17: Graph-based Methods for Natural Language Processing*, pages 19–39, 2024.

Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.

Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, volume 2, pages 119–129, 2000.

Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. Velvet: a novel ensemble learning approach to automatically locate vulnerable statements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 959–970. IEEE, 2022.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1729–1741. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00038. URL `https://doi.org/10.1109/ICSE55347.2025.00038`.

Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10507–10521, 2024.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pages 508–512, 2020.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547, 2020.

Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th international conference on mining software repositories*, pages 608–620, 2022.

Bao Guo, Chunxia Zhang, Junmin Liu, and Xiaoyi Ma. Improving text classification with weighted word embeddings via a multi-channel textcnn model. *Neurocomputing*, 363:366–374, 2019.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2022.

David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

Zhonghao Jiang, Weifeng Sun, Xiaoyan Gu, Jiaxin Wu, Tao Wen, Haibo Hu, and Meng Yan. Dfept: data flow embedding for enhancing pre-trained model based vulnerability detection. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pages 95–104, 2024.

Joern Contributors. Joern: The bug hunter's workbench. `https://joern.io`, 2026.

Hongyu Kuang, Feng Yang, Long Zhang, Gaigai Tang, and Lin Yang. Leveraging user-defined identifiers for counterfactual data generation in source code vulnerability detection. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 143–150. IEEE, 2023.

Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

Kyung-Suk Lhee and Steve J Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: practice and experience*, 33(5):423–460, 2003.

Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 292–303, 2021.

Junliang Liu, Jingyu Xiao, Wenxin Tang, Zhixian Wang, Zipeng Xie, Wenxuan Wang, Minrui Zhang, and Shuanghe Yu. Benchmarking mllm-based web understanding: Reasoning, robustness and safety. *arXiv preprint arXiv:2509.21782*, 2025.

Shengchao Liu, Weili Nie, Chengpeng Wang, Jiarui Lu, Zhuoran Qiao, Ling Liu, Jian Tang, Chaowei Xiao, and Animashree Anandkumar. Multi-modal molecule structure–text model for text-based retrieval and editing. *Nature Machine Intelligence*, 5(12):1447–1457, 2023.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. Pre-training by predicting program dependencies for vulnerability analysis tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

Wang Luo, Di Wu, Hengyuan Na, Yinlin Zhu, Miao Hu, and Guocong Quan. Rethinking multimodal point cloud completion: A completion-by-correction perspective. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 7680–7688, 2026.

Van Nguyen, Trung Le, Olivier De Vel, Paul Montague, John Grundy, and Dinh Phung. Information-theoretic source code vulnerability highlighting. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

OpenAI. Codex. `https://github.com/openai/codex`, 2025.

OpenAI. Introducing GPT-5.5. `https://openai.com/index/introducing-gpt-5-5/`, 2026.

Tao Peng, Shixu Chen, Fei Zhu, Junwei Tang, Junping Liu, and Xinrong Hu. Ptlvd: Program slicing and transformer-based line-level vulnerability detection system. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 162–173. IEEE, 2023.

Qwen Team. Qwen3.6-35B-A3B. `https://huggingface.co/Qwen/Qwen3.6-35B-A3B`, 2026.

Donald Ray and Jay Ligatti. Defining code-injection attacks. *Acm Sigplan Notices*, 47(1):179–190, 2012.

Rijha Safdar, Danyail Mateen, Syed Taha Ali, Umer Ashfaq, and Wajahat Hussain. Data and context matter: Towards generalizing ai-based software vulnerability detection. *International Journal of Information Security*, 25(5):145, 2026.

Mario Calín Sánchez, Juan Manuel Carrillo de Gea, José Luis Fernández-Alemán, Jesús Garcerán, and Ambrosio Toval. Software vulnerabilities overview: A descriptive study. *Tsinghua Science and Technology*, 25(2):270–280, 2019.

Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

Wenxin Tang, Jingyu Xiao, Wenxuan Jiang, Xi Xiao, Yuhang Wang, Xuxin Tang, Qing Li, Yuehe Ma, Junliang Liu, Shisong Tang, et al. Slidecoder: Layout-aware rag-enhanced hierarchical slide generation from design. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 9026–9050, 2025.

Wenxin Tang, Jingyu Xiao, Yanpei Gong, Fengyuan Ran, Tongchuan Xia, Junliang Liu, Man Ho Lam, Wenxuan Wang, and Michael R Lyu. Efficientpostergen: Semantic-aware efficient poster generation via token compression and accurate violation detection. *arXiv preprint arXiv:2603.00155*, 2026.

Chung-Nan Tsai, Xin Wang, Cheng-Hsiung Lee, and Ching-Sheng Lin. A sequential multi-stage approach for code vulnerability detection via confidence- and collaboration-based decision making. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 21151–21157, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1071. URL `https://aclanthology.org/2025.emnlp-main.1071/`.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–357. IEEE, 2023.

Xin-Cheng Wen, Cuiyun Gao, Shuzheng Gao, Yang Xiao, and Michael R Lyu. Scale: Constructing structured natural language comment trees for software vulnerability detection. In *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis*, pages 235–247, 2024.

Cheng Weng, Yihao Qin, Bo Lin, Pei Liu, and Liqian Chen. Matsvd: Boosting statement-level vulnerability detection via dependency-based attention. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pages 115–124, 2024.

Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, Hong Jin Kang, and David Lo. Let the trial begin: A mock-court approach to vulnerability detection using llm-based agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 1030–1042, 2026.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 241–253. IEEE, 2025a.

Jingyu Xiao, Ming Wang, Man Ho Lam, Yuxuan Wan, Junliang Liu, Yintong Huo, and Michael R Lyu. Designbench: A comprehensive benchmark for mllm-based front-end code generation. *arXiv preprint arXiv:2506.06251*, 2025b.

Jingyu Xiao, Jiantong Qin, Shuoqi Li, Man Ho Lam, Yuxuan Wan, Jen-tse Huang, Yintong Huo, and Michael R Lyu. Component-based reusable ui code generation for complex websites via semantic segmentation and fine-grained feedback. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 5638–5649, 2026a.

Jingyu Xiao, Zhongyi Zhang, Haoran Hou, Yuxuan Wan, Yuan Jiang, Yintong Huo, and Michael R Lyu. Visualrepair: Dynamic tool calling and region focusing for visual software issue repair. *arXiv preprint arXiv:2607.14075*, 2026b.

Jingyu Xiao, Zhongyi Zhang, Yuxuan Wan, Yintong Huo, Yang Liu, and Michael R Lyu. Efficientui-coder: A bidirectional token compression framework for efficient mllm-based ui code generation. *Proceedings of the ACM on Software Engineering*, 3(FSE):2396–2418, 2026c.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.

Shenghao Ye, Yu Guo, Dong Jin, Yuxiang Wang, Yikai Shen, Yunpeng Hou, Shuangwu Chen, Xiaofeng Jiang, et al. When tableqa meets noise: A dual denoising framework for complex questions and large-scale tables. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24022–24045, 2026a.

Shenghao Ye, Yu Guo, Zhengheng Li, Qiang Huang, Shuangwu Chen, Wang Luo, Rui Qian, Jie Zhang, Tao Zhang, Dong Jin, et al. Rubric-guided process reward for stepwise model routing. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing*, pages 27892–27910, 2026b.

Shenghao Ye, Yuxiang Wang, Yu Guo, Tao Zhang, Shuangwu Chen, Zijian Wen, Qihui Zhu, Mengjie Zhang, Dong Jin, Wang Luo, et al. Rethinking stepwise model routing: A cost-efficient table reasoning perspective. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing*, pages 28102–28120, 2026c.

Yue Yu, Wang Luo, Shenghao Ye, Zepeng Bao, Jiacheng Liu, Yongqi Li, Wenxin Tang, and Tieyun Qian. Format as nuisance: Intervention-inspired cross-format distillation for knowledge conflicts. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing*, pages 46190–46211, 2026.

Z.ai. GLM-4.7-Flash. `https://huggingface.co/zai-org/GLM-4.7-Flash`, 2026.

Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering*, 49(8): 4196–4212, 2023.

Qingsong Zou, Jingyu Xiao, Qing Li, Zhi Yan, Yuhang Wang, Li Xu, Wenxuan Wang, Kuofeng Gao, Ruoyu Li, and Yong Jiang. Queryattack: Jailbreaking aligned large language models using structured non-natural query language. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 5725–5741, 2025.

A Limitations

While DCVD achieves substantial improvements over existing methods, several limitations remain. First, the semantic branch relies on an offline LLM to generate natural language explanations of code, which introduces API costs and sensitivity to prompt engineering; end-to-end learnable explanation modules could mitigate this dependency. Second, our evaluation is conducted only on C/C++ vulnerability datasets; the generalization of DCVD to other programming languages (e.g., Java, Python, Go) and other benchmarks (e.g., Devign, DiverseVul) has not yet been fully validated. Third, the structure branch depends on static analysis tools to extract ASTs and CFGs, and therefore

degrades when the input is an incomplete code fragment (e.g., a GitHub diff hunk or a partially visible function in an IDE) for which static analyzers fail to produce reliable graphs; more robust structural representations or integration with pre-trained dependency predictors may help alleviate this issue.

B Relation to Graph–Text Cross-Modal Learning

Outside computer security, several studies model cross-modal relationships between graph structures and textual semantics. MoleculeSTM (Liu et al., 2023) constructs cross-modal pairs from the two-dimensional structure graph of a molecule and its natural-language description, and maps them into a shared space through contrastive learning, supporting structure–text retrieval and text-guided molecule editing. ConGraT (Brannon et al., 2024) pairs the graph context of a node with the document associated with that node in text-attributed graphs, and learns joint representations with a CLIP-style contrastive objective, supporting node classification, text classification, and link prediction. These studies show that when a graph and a text describe the same object from different perspectives, their co-occurrence can be turned into effective joint representations through cross-modal alignment.

Vulnerabilities exhibit the same structural–semantic character. Code tokens and security-oriented explanations convey semantic information about input processing, boundary checks, and functional intent, whereas ASTs and CFGs represent the syntactic hierarchy, branch conditions, and execution paths in which these operations occur. A semantically risky operation alone does not necessarily constitute a vulnerability; a genuine vulnerability pattern arises only when such an operation co-occurs with a missing check, a specific branch reachability, or an unsafe execution path. Conversely, structural information alone cannot determine the security implications of the relevant nodes in the current context.

Building on this co-occurrence principle, DCVD makes three task-specific adaptations. First, it replaces generic object graphs with ASTs and CFGs that carry program semantics, encoded by two independent GATs to model syntactic hierarchies and execution flows. Second, it introduces bidirectional structure–semantic cross-attention, which refines global modal alignment into interactions between graph nodes and code tokens before fusion. Third, it jointly optimizes function-level detection and explicitly supervised statement-level localization, so that the cross-modal representation supports both holistic prediction and fine-grained localization. The fusion ablation in Section 4.3 supports this adaptation: replacing contrastive alignment and bidirectional cross-attention with concatenation and an MLP lowers function-level F1 from 94.05 to 84.02, One-Phase statement-level F1 from 87.81 to 51.23, and Top-1 from 93.12 to 79.35.

C Implementation Details

C.1 Configuration Parameters

We summarize the configuration of DCVD in Table 6.

Table 6: Configuration of DCVD.

Category	Parameter	Value
Model Architecture	d' (branch / fusion dimension)	128
	d_k (deep representation dimension)	256
	GAT layers	2
	Self-attention heads (statement branch)	8
	Max sequence length	512
Training	Optimizer	AdamW
	Learning rate	2×10^{-5}
	LR scheduler	Cosine with restarts
	Warmup steps	500
	Batch size	32
	Epochs	50
	α (loss balancing)	0.4

C.2 Runtime Environment and Training Time

DCVD is implemented in PyTorch with PyTorch Geometric for graph operations, and is trained using HuggingFace Transformers and the Accelerate library. All experiments are conducted on NVIDIA H20 GPUs, and one complete training run takes approximately 20 hours.

D Stability and Significance of the Main Results

We independently run DCVD and the learning-based baselines with the same five random seeds, run each LLM-based baseline five times independently, report the standard deviation of DCVD and the strongest baseline on the main metrics, and conduct two-sided Welch's t-tests between DCVD and the strongest baseline for each metric. Holm correction is applied across the nine comparisons on the three datasets. Table 7 shows that the standard deviations are small and that all improvements remain significant after correction, indicating that the gains of DCVD are stable across random seeds and independent runs.

Table 7: Standard deviation of DCVD and the strongest baseline over five independent runs and Holm-adjusted p-values of two-sided Welch's t-tests against the strongest baseline. *, **, and *** denote $p < 0.05$, $p < 0.01$, and $p < 0.001$.

Dataset	Setting	Metric	DCVD Std.	Strongest Baseline	Baseline Std.	Adjusted p
BigVul	Func-Level	Mcc	0.0070	MatsVD	0.0042	0.0407^{*}
BigVul	Func-Level	F1	0.57	MatsVD	0.41	0.0407^{*}
BigVul	Two-Phase Stmt-Level	Mcc	0.0074	MatsVD	0.0068	0.0063^{**}
BigVul	Two-Phase Stmt-Level	F1	0.71	MatsVD	0.64	0.0063^{**}
BigVul	One-Phase Stmt-Level	Mcc	0.0080	MatsVD	0.0055	0.0407^{*}
BigVul	One-Phase Stmt-Level	F1	0.65	MatsVD	0.50	0.0407^{*}
PrimeVul	Func-Level	F1	1.08	ConColl	1.37	$<0.001^{***}$
PrimeVul	Func-Level	Acc	1.16	Codex + GPT-5.5	1.29	$<0.001^{***}$
VulGate	Func-Level	F1	0.85	VulTrial	0.62	0.0096^{**}

E Significance Analysis of Ablation Study

To verify that the improvements reported in the ablation study are not due to random fluctuations, we further conduct a significance analysis. Specifically, for each variant we train the model three times with different random seeds and evaluate each run on the 30% sampled test set, and perform a paired t-test against the full DCVD on both Two-Phase Function-Level F1 and One-Phase Statement-Level F1. The resulting p-values are reported in Table 8, where ***, **, and * denote $p < 0.001$, $p < 0.01$, and $p < 0.05$ respectively, and "n.s." indicates non-significance ($p \geq 0.05$).

Table 8: Paired t-test p-values of each ablation variant compared with the full DCVD.

Variant	Func F1 (p-value)	Stmt F1 (p-value)
w/o Structure	$< 0.05^{*}$	$< 0.01^{**}$
w/o Semantic	$< 0.05^{*}$	$< 0.05^{*}$
w/o Fusion	$< 0.001^{***}$	$< 0.001^{***}$
w/o Statement-Level Task	$< 0.001^{***}$	–
w/o Function-Level Task	–	$< 0.01^{**}$

Overall, all eight comparisons yield statistically significant differences, confirming that the contributions of the Cross-Modal Fusion module and the Multi-Granularity Supervisor, as well as the overall design of the Dual-Channel Encoder, are robust rather than coincidental.

F Robustness to the Explanation LLM

Previous studies have highlighted the importance of evaluating multimodal models across diverse tasks and input conditions (Xiao et al., 2025b, 2026b). To measure the sensitivity of DCVD to the

explanation model, we replace GPT-4o-mini in the semantic branch with two locally deployable open-source Mixture-of-Experts models, Qwen3.6-35B-A3B (Qwen Team, 2026) and GLM-4.7-Flash (Z.ai, 2026), which activate about 3B parameters per token, and evaluate function-level detection on PrimeVul. As Table 9 shows, F1 changes by at most 1.51 points across the three models, and both open-source variants still outperform all baselines in Table 3 in F1 and Acc. The explanation LLM describes function behavior, code intent, security-relevant operations, and directly observable risk indicators, rather than directly providing the final vulnerability prediction. Future work could explore cost-aware routing (Ye et al., 2026c) and rubric-guided process rewards (Ye et al., 2026b) to further balance the computational cost and quality of explanation generation.

Table 9: Function-level detection results of DCVD on PrimeVul with different semantic explanation models.

Explanation LLM	Pre	Re	FPR↓	F1	Acc
GPT-4o-mini	**70.27**	**71.72**	**30.34**	**70.99**	**70.69**
Qwen3.6-35B-A3B	69.82	71.26	30.80	70.53	70.23
GLM-4.7-Flash	68.85	70.11	31.72	69.48	69.20

G Broader Impacts

This work aims to improve automated software vulnerability detection and localization, with the primary goal of assisting developers, security analysts, and maintainers in identifying security-critical code regions more accurately and efficiently. By jointly predicting whether a function is vulnerable and localizing the responsible statements, DCVD can potentially reduce the manual effort required for code auditing, accelerate vulnerability triage, and support the development of more secure software systems. These benefits are particularly relevant for large-scale open-source and industrial codebases, where manual security review is often costly and difficult to scale.

At the same time, vulnerability detection and localization techniques may have potential dual-use risks. In particular, fine-grained localization results could be misused by malicious actors to identify exploitable code regions more efficiently. In addition, incorrect predictions may introduce practical risks: false negatives may leave vulnerabilities undetected, while false positives may increase the workload of developers and security teams. Therefore, DCVD should be used as an assistive tool for defensive security analysis rather than as a replacement for expert review, manual validation, or secure development practices.

To mitigate these risks, we recommend that released artifacts and experimental results be used only for defensive security research, vulnerability mitigation, and responsible software maintenance. When deployed in real-world auditing workflows, predictions should be reviewed by qualified security practitioners before any security-critical decision is made. We also encourage future work to incorporate uncertainty estimation, human-in-the-loop validation, and responsible disclosure mechanisms to further reduce the risk of misuse and improve the reliability of automated vulnerability analysis.

H Prompt Template

This section provides the prompt template used by the LLM-based semantic branch in DCVD and the prompting protocol of the LLM-based baselines. The placeholder enclosed by angle brackets is replaced with the corresponding input code during inference.

H.1 Semantic Branch: Code Explanation Generation

The semantic branch invokes an LLM to generate a natural-language explanation of the input code. The generated explanation is used to capture functional intent, behavioral logic, API semantics, and security-relevant behaviors that are difficult to obtain from structural analysis alone. More broadly, denoising complex inputs offers a complementary perspective on isolating task-relevant information for reasoning (Ye et al., 2026a).

Prompt Template for Code Explanation Generation

```
You are an expert software security and program analysis assistant.

Given the following source code, generate a concise natural-language explanation
     of its functionality and potential security-relevant behaviors. The
    explanation will be used as semantic input for a vulnerability detection and
     localization model, so it should focus on observable program behavior
    rather than unsupported speculation.

Please analyze the code from the following aspects:

1. Functionality:
Describe the main purpose of the function and what task it performs.

2. Inputs and Outputs:
Identify important parameters, external inputs, return values, and output
    behaviors.

3. Control Logic:
Describe key branches, loops, conditions, and execution paths that affect the
    behavior of the function.

4. Data Flow:
Describe how important data values propagate through the function, especially
    from inputs to memory operations, pointer operations, array accesses, file
    operations, system calls, or other sensitive operations.

5. Security-Relevant Behaviors:
Identify potential security-relevant operations, such as boundary checks, null
    checks, memory allocation or release, pointer dereference, buffer access,
    arithmetic computation, authentication checks, and error handling.

6. Potential Risk Indicators:
Mention suspicious or risky behaviors only if they are directly observable from
    the code. Do not overclaim vulnerability existence.

Source Code:
<Code>

Output format:
Functionality: <brief description>
Inputs and Outputs: <brief description>
Control Logic: <brief description>
Data Flow: <brief description>
Security-Relevant Behaviors: <brief description>
Potential Risk Indicators: <brief description or N/A>
```

H.2 Prompts of the LLM-Based Baselines

VulTrial (Widyasari et al., 2026) and ConColl (Tsai et al., 2025) use the prompts released with their original papers without modification. GPT-5.5 and Codex + GPT-5.5 receive the prompt below, which elicits step-by-step reasoning and prohibits network retrieval so that the models cannot search online for the answers to the test samples. The predicted label is used for function-level evaluation, and the line scores are used for statement-level classification and ranking on BigVul.

Prompt Template for GPT-5.5 and Codex + GPT-5.5

```
You are an expert software security auditor.
```

```
Analyze the following C/C++ function and determine whether it contains a
    security vulnerability. Do not search the Internet or use any external
    resources; base your analysis only on the given code.

Please reason step by step:

1. Summarize the purpose of the function and identify its inputs.

2. Trace how external inputs and important variables flow through the function,
    paying attention to memory operations, pointer arithmetic, array indexing,
    integer arithmetic, resource allocation and release, and error handling.

3. Check whether the necessary boundary checks, null checks, and validity checks
     are present on every execution path.

4. Decide whether the function is vulnerable.

5. Assign every line a score in [0, 1] indicating how likely it is to be
    responsible for the vulnerability.

Source Code (each line is prefixed with its line number):
<Code>

Output format:
Reasoning: <step-by-step analysis>
Vulnerable: <YES or NO>
Line Scores: <line number>: <score>, one entry per line, sorted by score in
    descending order
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction accurately reflect the contributions and scope of the paper.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations of our work in Appendix A.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.

- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: Our paper does not involve theoretical result.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We provide a detailed description of the method in Section 3, include the code in the Supplementary Material, and present the detailed configuration parameters in C.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.

- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide the code in the Supplementary Material.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: We describe this in Section C.1.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We provide statistical significance analysis for the ablation study in Section D and Section E.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: We describe the compute resources in Section C.2.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: We have reviewed and complied with the NeurIPS Code of Ethics.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: This work may have positive societal impacts by improving automated vulnerability auditing and helping developers identify security risks. It may also have potential dual-use risks because fine-grained localization results could be misused to identify exploitable code regions. We discuss these impacts and mitigation considerations in Appendix G.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: This paper poses no such risks.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We cite all external datasets and models used in this work, and provide their corresponding licenses and access terms in the README file of the supplementary material.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide the code with documentation in the supplementary material.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Our methodology involves using an LLM to provide explanations for the input code. The specific LLM used in our experiments is reported in Section 4.1.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.