Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.082481

ARTICLE

Large Language Model-Based Representations of Heterogeneous Graphs for Vulnerability Detection

Xiaorong Feng[1,2], **Ying Gao**[1,*] **and Leyu Shi**[2]

[1]School of Computer Science & Engineering, South China University of Technology, Guangzhou, China
[2]China Electronic Product Reliability and Environmental Testing Research Institute, Guangzhou, China

*Corresponding Author: Ying Gao. Email: gaoying@scut.edu.cn

Received: 17 March 2026; Accepted: 22 May 2026; Published: 23 July 2026

ABSTRACT: Open source software has become a fundamental component of modern software ecosystems, supporting a wide range of critical applications in operating systems, cloud services, embedded systems, and security-sensitive infrastructures. However, the rapid growth of open source projects also brings increasingly serious security challenges. Many widely used C/C++ components still contain hidden vulnerabilities, and attackers are no longer limited to exploiting traditional memory-related bugs such as buffer overflows or use-after-free errors. In recent years, non-memory logic flaws, including improper authentication, incorrect state transitions, flawed boundary checks, and insecure API usage, have become more prevalent and more difficult to detect using conventional static analysis or pattern-matching methods. To address these limitations, this study proposes a novel vulnerability detection framework that combines the semantic understanding capability of large language models (LLMs) with the structural representation ability of heterogeneous graph learning. Specifically, we construct a Heterogeneous Vulnerability Graph (HeVG) to explicitly model multiple types of code structures in C/C++ programs, including syntax, control flow, data dependency, and function-call relationships. By representing source code as a heterogeneous graph, the proposed framework can capture both local code patterns and long-range dependencies that are essential for identifying complex vulnerabilities. In addition, a cross-modal alignment mechanism is introduced to effectively fuse code-text semantic features extracted by LLMs with graph-based structural representations. This enables the model to jointly understand what the code means and how different program elements interact. Experimental results show that the proposed approach achieves state-of-the-art performance, reaching 95.61% accuracy in single-file vulnerability detection and 90.97% accuracy in cross-file vulnerability detection. Further analysis demonstrates that the framework is particularly effective in detecting complex logic vulnerabilities and maintains strong generalization ability across different projects. These results indicate that integrating LLMs with heterogeneous graph learning provides a promising direction for more accurate and robust open source software vulnerability detection. Open source software faces growing security challenges, with widespread vulnerabilities in critical components and an increasing prevalence of non-memory logic flaws. To address these issues, this study proposes a novel vulnerability detection framework that integrates large language models (LLMs) with heterogeneous graph learning. We introduce a Heterogeneous Vulnerability Graph (HeVG) to explicitly model diverse code structures in C/C++ programs, and employ a cross-modal alignment mechanism to fuse semantic information from code text with graph representations. Experimental results demonstrate the effectiveness of our approach, achieving state-of-the-art performance in both single-file (95.61% accuracy) and cross-file (90.97% accuracy) vulnerability detection. The framework shows particular strength in identifying complex logic vulnerabilities while maintaining high generalization capability across projects.

KEYWORDS: Large language model; graph neural network; open source software security; vulnerability detection; heterogeneous vulnerability graph

1 Introduction

Since the explosive growth of open-source software has brought acute challenges to software security and open-source software now underlies more than 70% of enterprise projects [1], it is natural and important for automated vulnerability detection to aim at building a model that can reliably find subtle defects in complex code-bases without being misled by the semantic gaps inherent in compiled representations. Hence, the literature on this subject can reasonably be organized around three distinct detection paradigms, each with its own approach to interpreting code logic. Due to the fact that initial paradigm may properly be called passive signature matching, it is natural to say that detection is regarded by it as a retrieval procedure. Illustrated by conventional CVE/NVD-based methods (Common Vulnerabilities and Exposures/National Vulnerability Database-based methods), these approaches depend on recognized patterns and have difficulty in adapting to the shrinking window of zero-day exploits. Apart from this, it lies the paradigm of logic-rule static examination, in which tools utilize set rules to search for flaws. However, these approaches frequently face difficulties when dealing with non-memory-corrupting logical glitches, such as race conditions and authentication bypasses, which currently make up 47% of vulnerabilities, and have challenges handling the non-deterministic behavior brought by cloud-native multi-threading. The third paradigm depends on uniform representation learning. Methods using RNNs or regular graph neural networks (GNNs) transform code into one-dimensional sequences, attaining lessthan-optimal outcomes in capturing long-distance dependencies (F1 < 0.5) and lacking uniformity across architectures such as Acorn Reduced Instruction Set Computer Machine (ARM) and Reduced Instruction Set Computer Five (RISC-V) [2,3]. However, current methods frequently struggle when faced with the non-stationary distribution changes that are characteristic of real-world exploitation situations. The restrictions mostly argued with two aspects. Firstly, the semantic disparity in de-compilation. Most present approaches trust middle-level representations without question, while around 35% of control-flow edges in the pseudo-code produced by tools such as IDA Pro might not represent actual execution paths. This results in a recursive drift in which the model constructs its logic based on faulty structural assumptions. Secondly, the uncertainty of concurrency within weak memory models. Current tools frequently struggle to tell real race conditions apart from harmless noise, leading to a 37% rise in false positives on architectures such as ARM. Uncritically examining these structures stops the model from internalizing unchanging laws, making it focus on hallucinations (false correlations) instead of long-lasting vulnerability logic. This disparity prompts a basic question: how can a model overcome the curse of structural ambiguity and attain profound understanding? We draw motivation from the cognitive mechanism of human security specialists. In manual auditing, this issue is tackled by the Cognitive Bifurcation that occurs between Structural Sensing and Semantic Reasoning. The specialists gaze rapidly follows the physical control flow to find potential risk spots, while their knowledge repository deduces the high-level purpose behind unclear function calls. Importantly, genuine detection resilience stems from Cross-Modal Resonance. The abstract semantic comprehension corrects the ambiguity of the structural graph, making sure that local syntactic variations do not hide the global vulnerability pattern. Guided by this insight, we put forward a framework that regards vulnerability detection as a cooperative process between diverse structural modeling and large language model (LLM) reasoning, namely HeVG-LLM. Our framework presents two interwoven innovations: First, to alleviate structural ambiguity, we devise a Heterogeneous Vulnerability Graph (HeVG) representation. Unlike uniform methods, HeVG clearly models the variety of code elements (e.g., API invocations, pointer calculations) and edge categories. By re-constructing the anatomy of program running with detailed distinctions, this structural framework offers an accurate physical basis for the model, efficiently separating relevant dependency chains from non-relevant syntax interference. Second, to bridge the semantic gap, we introduce a Text-Graph-Cross-Modality Alignment mechanism. Rather than handling the graph and text as distinct modalities, we adopt a fusion approach in which semantically enhanced embeddings.

It was produced by HiGPT with a code-specialized LLM that are thoroughly incorporated with the graph structure. This alignment makes sure that the model not merely captures the local dependencies but also leverages global semantic insights to reason regarding complex, cross-file logic flaws. Our contributions are summarized as follows:

- We propose a **Heterogeneous Vulnerability Graph (HeVG)** representation that explicitly models diverse code semantics and structural dependencies, overcoming the limitations of homogeneous graphs [4].
- An **LLM-based Cross-Modal Alignment** approach is presented where the semantic embeddings obtained from HiGPT are naturally and properly incorporated into graph structural features. The proposed alignment can smoothly bridge the gap between low-level code and high-level intent item.
- Since we have implemented a Multi-Granularity Detection method whose purpose is to expand the heterogeneous graph so that cross-file context can be used to detect behavior, it is natural that the project-level analysis yields much higher accuracy.
- From the available experimental results it is clearly and convincingly established that the proposed framework achieves state-of-the-art performance, attaining 95.61% accuracy for single-file cases and generalizing well for cross-file situations.

2 Related Work

Since the combined approach is being proposed, it is natural and appropriate to discuss three major issues relevant to vulnerability identification in the related work section. In terms of semantic gap bridging, traditional methods have a semantic discontinuity in code de-compiling. Cosine similarity between semantics on intermediate representations and the source code is frequently less than 0.3. As the Heterogeneous Vulnerability Graph (HeVG) was capable of modeling those elements of the code. They were explicitly typed as nodes and edges and appropriately coupled with HiGPT-generated semantically-rich nodes embeddings. The generated HeVG provides a gap between representation level semantics with low level and high level program semantics. Besides the semantic gap, other mitigating factor to concurrency uncertainty is the mitigation of concurrency uncertainty. Even now problematic issues of detection due to thread interactions and scheduling anomalies still exist. To explicitly represent such elements of concurrency-relevance controls, the HeVG includes threads, locks and shared variables as various node types. The Heterogeneous Graph Transformer (HGT) is then used to model the temporal dynamics using heterogeneous attention processes. Besides the next two factors, we must be able to overcome feature representation bottlenecks. Lack of ability to represent long-range dependencies and interpolating over recurrent neural networks (RNNs) illustrate in structures such as regressive sequence models. The cross-architecture consistency can be reduced by less than 68%. We synthesizes multi-source heterogeneous data by using a hybrid LLM-graph network, and this may obtain breakthrough performance. The single file detection will be able to do 95.61% and the cross file will be able to do 90.97% precision, and maintains 89% consistency between ARM and RISC-V. In addition, recent studies further motivate this line of research from the perspective of LLM-based software security analysis, knowledge-graph-guided vulnerability reasoning, Code Property Graph (CPG)-guided large language models, and the combination of HGTs with LLMs [5–9].

2.1 Feature-Based Vulnerability Detection Methods

One of the major transformations that change methods of rule to more advanced data-driven ones has been employed in binary code. Some tools like FlawFinder which were based on the regular expressions defined by experts could be mostly burdened by the false-positive rates that were high and frequently above 40 percent. Data-driven techniques became the standard due to the introduction of deep learning that

has since prompted paradigm shift. The fundamental concept of these techniques is the automated feature learning in order to obtain accurate and scalable vulnerability detection. Multi-stage feature engineering is a framework that focuses on deep learning pipeline. It starts with code pre-processing, where normalization to mitigate semantic noise from varied coding styles and slicing to isolate vulnerability-relevant data and control flow paths is implemented. A structured graph representation like Code Property Graph (CPG) is converted to model complex program dependencies by processed code. Subsequently, semantically rich vectors embed these graphs. Pre-trained Language Models (PLMs) such as CodeBERT [10] have shown significant advantages over traditional. Detection models training with these final vector representations, often enhanced with advanced strategies such as adversarial training for robustness or temporal networks for analyzing concurrent programs. Recent studies have further shown that advanced open-weight LLMs, such as Llama-3 and DeepSeek-Coder, can become competitive in software security analysis after task-specific fine-tuning, which provides stronger support for introducing modern LLMs into vulnerability detection pipelines [5,11]. In Recommendations as Treatments: Debiasing Learning and Evaluation [12], Schnabel et al. discuss how biased and incomplete observations can affect model learning and evaluation. This idea is relevant to our discussion because feature-centered vulnerability detection methods may also learn from incomplete structural contexts after code slicing, resulting in biased representations and missed vulnerabilities. Second, most existing models assume a single-threaded execution setting, thereby ignoring the non-deterministic behaviors in multi-threaded programs, which results in high false-positive rates (up to 52%) for concurrency bugs. Third, pre-trained language models (PLMs) do not adapt well to the idiosyncrasies of decompiled binary code, leading to a large semantic gap. Worse still, cross-platform architectures introduce additional, exacerbating differences. These limitations also explain why recent research has increasingly turned to structured guidance, rather than relying on pure sequence modeling alone, when applying LLMs to vulnerability localization and reasoning [6,7].

2.2 Graph-Based Vulnerability Detection Methods

HeVulD [13] is a clear and successful example of this approach applied to cross-module vulnerability detection. However, it is also evident that reliance on manually designed paths limits adaptability and may lead to the path explosion problem in complex concurrent programs. The present discussion concerns a subgraph-based learning approach which naturally extracts localized structures for feature acquisition, hence it is efficient and effective for some bugs, but also has a clear drawback: graph partitioning may disrupt important global dependencies such as cross-thread variable sharing. More recently, graph-guided LLM frameworks have extended this direction further. LLMxCPG demonstrates that a Code Property Graph can guide LLMs to focus on vulnerability-relevant code context while reducing the amount of code to be analyzed without losing key semantics [7]. VulReaD further shows that a security knowledge graph can serve as a semantic backbone for vulnerability reasoning and CWE classification, thereby improving interpretability and reducing unsupported reasoning [6]. In addition, MANDO-LLM combines Heterogeneous Graph Transformers with LLMs for smart contract vulnerability detection, making it one of the closest recent references to our work. However, unlike MANDO-LLM, which focuses on smart contracts, our method is designed for C/C++ vulnerability detection and emphasizes cross-file program structure and concurrency-aware semantics [8].

2.3 Application of Large Language Models (LLMs) in Code Analysis

Application of Large Language Models [14,15] (LLMs) in Code Analysis Large Language Models (LLM) have gained significant popularity in the code analysis community in the recent years, as they have high semantic understanding and generation abilities. GitHub Copilot, a product based on the Codex model,

provides automatic code completion, facilitating faster work of the developers. However, it has helped with binary reverseengineering, thus vulnerability discovery is limited. CodeT5 enhanced its comprehension of code semantics via multi-language pretraining. However, its adaptation to decompiled binary code is still less than ideal, unable to efficiently address the semantic deviation issue. Recent studies have further demonstrated that advanced LLMs can play a more direct role in software security analysis. In particular, comparative studies on fine-tuned Llama-3 indicate that modern open-weight models are becoming increasingly competitive with proprietary models in code-security tasks [5,11]. At the same time, recent work suggests that LLMs are more effective when guided by structured program or security knowledge. VulReaD employs a security knowledge graph to support vulnerability reasoning, while LLMxCPG uses Code Property Graph-guided context reduction to help LLMs handle long code inputs more effectively [6,7]. From a broader perspective, AERIS introduces a cognitive reasoning layer for dialectical evaluation and provides a useful theoretical perspective on how structured internal organization may improve model reasoning under conflicting evidence [9]. In contrast to the previously mentioned studies, the HiGPT model presented in this paper provides a new approach via a distinct graph-text alignment mechanism that allows for theprofound fusion of codes textual and structural features. By efficiently utilizing the textual details from code comments and API documentation, the model makes use of a multi-head attention mechanism to project these textual features onto the semantic depictions of graph nodes. This offers a novel and potent approach for vulnerability discovery in binary code. As shown on the Juliet dataset, this method cuts down the false positive rate by 37.3% when compared with the state-of-the-art HeVulD model.

3 Problem Statement

Static and dynamic analysis of intermediate representations (IR) or source code plays an important role in vulnerability detection. However, decompiled code, concurrent behaviors, and cross-architecture scenarios still introduce significant challenges, which expose the limitations of conventional homogeneous representations and shallow feature-learning models. To make the problem formulation more explicit, we characterize the vulnerability detection setting in this work by three concrete obstacles: semantic disparity, concurrency indeterminacy, and feature representation constraints. The purpose of introducing a heterogeneous graph is not merely to enrich representation, but to preserve semantically distinct program entities and relations that would otherwise be collapsed in a homogeneous graph. Likewise, the role of LLM-based text-graph alignment is not only to provide richer features, but to reduce the mismatch between low-level structural representations and higher-level semantic cues.

(1) Semantic Disparity Problem: The semantic disparity between decompiled code and source code is essentially a discontinuity between low-level IR and high-level semantics. Although pre-trained language models such as CodeBERT perform well on source code, their adaptation to decompiled code is often limited because decompiled programs contain redundant variables, irregular structures, and imprecise types. In this work, we model this issue as a representation mismatch between structural graph features and textual/semantic cues attached to code elements. To mitigate this mismatch, we introduce a text-graph alignment mechanism that projects node-associated text embeddings into the graph representation space and fuses them through cross-modal attention. This makes the claim of "bridging semantic disparity" operational rather than purely conceptual.

(2) Concurrency Indeterminacy: The unpredictability of multi-threaded vulnerabilities stems from sparse, dynamic, and heterogeneous interactions among threads, locks, and shared variables. Traditional homogeneous program graphs simplify these interactions into a uniform structure and therefore cannot explicitly preserve the distinct semantics of synchronization-related entities and relations. In contrast, our HeVG explicitly represents concurrency-relevant entities as typed nodes and typed edges. Under this

formulation, concurrency indeterminacy is not assumed to be "solved" in the abstract; rather, it is addressed by retaining synchronization-related structure that is discarded in homogeneous representations. Combined with key-node slicing and GNN message passing, this design provides a concrete mechanism for propagating concurrency-relevant information across the graph.

(3) Feature Representation Constraint: Existing deep learning models such as RNNs have difficulty integrating long-range dependencies and multiple semantic views of code, including syntax, control flow, data flow, and type information. Therefore, a representation learning framework capable of integrating multi-source heterogeneous program information is needed. Heterogeneous graph models provide a natural basis for this purpose by unifying different code entities and relations in a single graph structure. Meanwhile, LLMs serve as semantic encoders for node-associated textual information, enabling the model to incorporate identifier names, API names, and comments into the structural representation. The resulting hybrid representation is designed to address feature bottlenecks by combining structured reasoning with semantic alignment.

Accordingly, the problem studied in this paper can be stated as follows: given a program P, construct a heterogeneous vulnerability graph $G = (\mathcal{V}, \mathcal{E})$ together with node-associated textual information $\mathcal{T}$, and learn a classifier $f(G, \mathcal{T}) \rightarrow y$, where $y \in \{0, 1\}$ indicates whether the program contains a vulnerability. Under this formulation, heterogeneous graph modeling is intended to preserve structurally distinct program dependencies, while LLM-based alignment is intended to reduce semantic mismatch between text and graph representations. Their effects are later examined empirically in the ablation studies and comparative experiments in Section 6.10.

In summary, semantic disparity, concurrency indeterminacy, and feature representation constraints constitute three major challenges in vulnerability detection. The proposed framework is motivated by the hypothesis that heterogeneous graph modeling and LLM-based alignment address different but complementary aspects of this problem: the former preserves typed structural dependencies, while the latter enriches and aligns semantic information.

4 Framework

The overall workflow of the proposed framework is illustrated in Fig. 1. For clarity, the pipeline is organized into three sequential stages. In the first stage (blue region), C/C++ source files are parsed to generate abstract syntax tree (AST) and control-flow graph (CFG) representations, from which the program dependence graph (PDG) is constructed. Based on the node and edge definitions introduced in Sections 4.1.1 and 4.1.2, the PDG is further transformed into the Heterogeneous Vulnerability Graph (HeVG), where key nodes related to API calls, array operations, pointer operations, and arithmetic operations are highlighted. In the second stage (green region), node-associated textual information, such as comments, identifiers, and API descriptions, is encoded into semantic vectors, while structural features are extracted from HeVG. These two modalities are then aligned through a multi-head cross-modal attention module to obtain fused node representations. In the third stage (yellow region), the fused node representations are processed by HGT layers to model heterogeneous dependencies, followed by hierarchical pooling to generate a graph-level representation. Finally, an MLP with a Softmax layer outputs the vulnerability prediction. The arrows in Fig. 1 indicate the data flow between these stages.

4.1 Heterogeneous Graph for Vulnerability Detection

As illustrated in Fig. 2, the HeVG construction process starts from the original code snippet and proceeds through node-type annotation on the AST/PDG representation. Data dependencies, control dependencies, and key nodes are then identified to generate the final HeVG subgraph. This figure provides

a concrete example of how the heterogeneous graph is derived from source code and how vulnerability-relevant structures are preserved during graph construction. Homogeneous graphs are in effect the predominant type of graph-based vulnerability detection tools tried, as they give no differentiation between different types of code statements and dependencies that they have, and thus, they cannot accurately represent source code semantics and structures. We are suggesting a solution to this shortcoming that is a Heterogeneous Vulnerability Graph (HeVG) specially designed to work with C/C++ code and exposes complexity of code in an explicit manner.

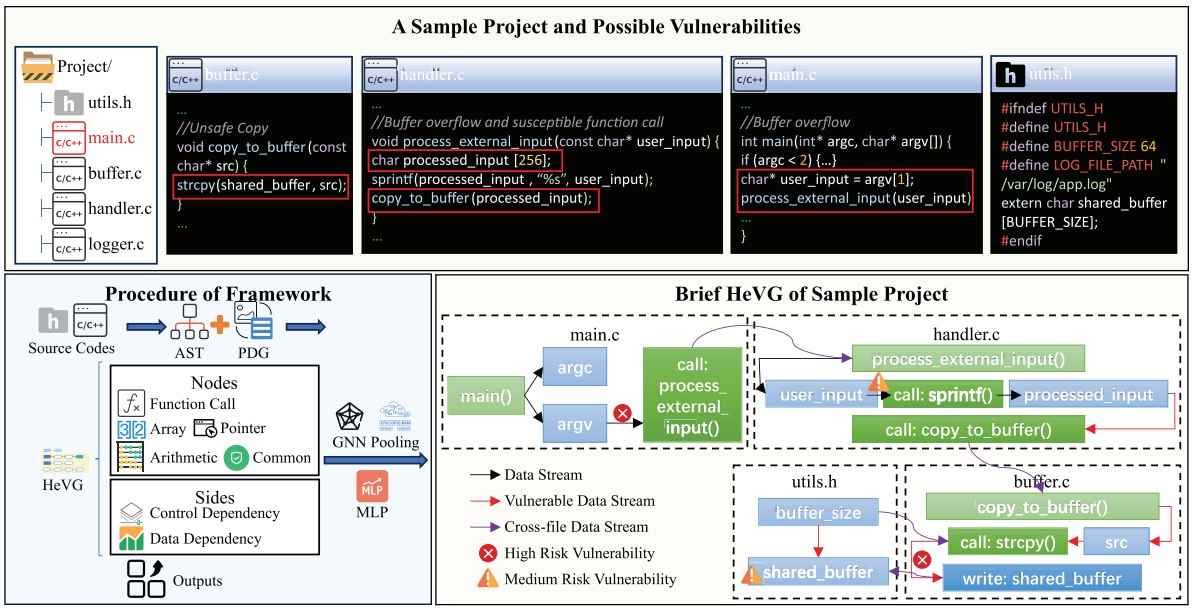

Figure 1: Overview of the proposed vulnerability detection framework. The pipeline consists of three color-coded stages: code parsing and HeVG construction (blue), multimodal feature extraction and text-graph alignment (green, with the alignment module highlighted in purple), and graph neural network-based vulnerability classification (yellow). Key nodes related to vulnerability-sensitive operations are highlighted in orange, and arrows indicate the data flow across stages.

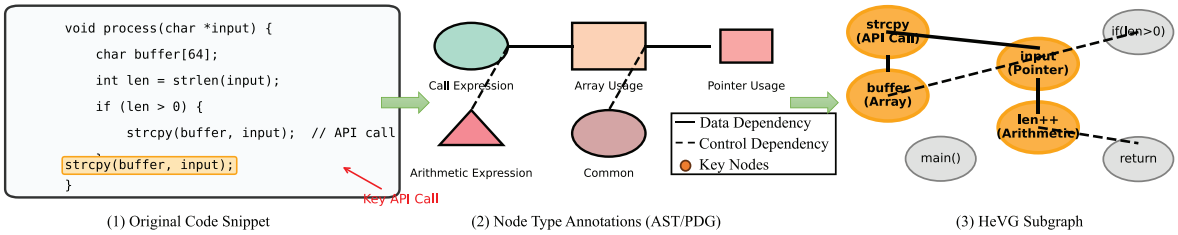

Figure 2: Illustration of the Heterogeneous Vulnerability Graph (HeVG) construction process. (1) An original vulnerable code snippet containing a strcpy call that may lead to buffer overflow; (2) the code is parsed into an AST/PDG representation with annotated node types (e.g., call expression, array usage); (3) key nodes (e.g., API calls, variables) and their data/control dependencies are extracted to build the corresponding HeVG subgraph.

The concept of HeVG is formally defined as the graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ denote the set of nodes and edges respectively. The heterogeneity is defined by two mapping functions: type of node in maps $\phi : \mathcal{V} \to \mathcal{A}$ (where $\mathcal{A}$ is the set of given node types) and type of node in maps $\psi : \mathcal{E} \to \mathcal{R}$ (where $\mathcal{R}$ is the set

of given edge types). A graph is said to be heterogeneous if $|\mathcal{A}| + |\mathcal{R}| > 2$, The existence of two or more of the types of nodes, edges, or a combination of both.

4.1.1 Node Type Definition

To build HeVG, we need to define the node types initially in two mutually complementary viewpoints. In the first one the nodes are classified according to core elements of C/C++ language yielding five category which include, **Call Expression** (statements which have function/method calls), **Array Usage** (statements which contain array operations), **Pointer Usage** (statements which contain pointer operations), **Arithmetic Expression** (statements containing arithmetic or bitwise operations) and **Common** (statements which do not fall into the category of the above). A code statement can have many nodes associated with it in case it is composed of many elements of the language (i.e., a statement that has both an array access and a function call will create two nodes). The second mode identifies nodes according to their functional roles and these include **Condition** (Boolean expressions in the control flow), **Expression** (computational statements), **Definition** (declarations or definitions) and **Common** (other statements). Moreover we define four types of key-nodes with **frequent API Function Call, Array Operation, Pointer Operation and Arithmetic Operation** because they are very relevant with vulnerability.

4.1.2 Edge Type Definition

There are two basic dependencies of the Program Dependence Graph (PDG) that HeVG preserves in its edge types CDE (Control Dependency Edges) and DDE (DataDependency Edges). There is a from statement S_1 to S_2 exists when the performance of S_2 is based on the performance of S_1. S_2 is formally control-dependent on S_1 when there is a path between S_1 and S_2 all of the intermediate nodes of which must lead to S_2, but S_1 need not lead to S_2. The existence of a DDE between S_1 and S_2 from S_1 data being used by S_2 means:

$$[I(S_1) \cap O(S_2)] \cup [O(S_1) \cap I(S_2)] \cup [O(S_1) \cap O(S_2)] \neq \varnothing \tag{1}$$

where $I(S_i)$ is the set of memory locations read by S_i, and $O(S_j)$ is the set of memory locations written by S_j.

4.1.3 Cross-File Extension

To support the analysis of vulnerabilities spanning multiple files, we extend HeVG to include **inter-file edges**. Specifically, we define:

- **CallsToExternal:** The name given to an edge from a *Call Expression* node in file F_i to a function definition node in file F_j ($i \neq j$).
- **SharesGlobal:** Defination of an edge which connects two nodes (e.g., a variable definition and its usage) that reference the same global variable defined in a third file.

Due to these edges enable the GNN to propagate information across file boundaries, they provide a natural way to model project-level dependencies, as is demonstrated in the cross-file experiments (Section 6.5).

4.1.4 Graph Construction though Key-Node Slicing

HeVG is built through key-node-based program slicing in order to target code regions of relevance to vulnerability and minimize redundancy, as summarized in Algorithm 1. For a specified key-node, forward and backward scans on the updated PDG (heterogeneous nodes and edges) are carried out. Let $\mathcal{V}_f$ and $\mathcal{E}_f$ be

the nodes and edges collected in the forward traversal (starting from the key-node), and $\mathcal{V}_b$ and $\mathcal{E}_b$ be those from the backward traversal (ending at the key-node). The final HeVG is the union of these traversal results:

$$\mathcal{V}_h = \mathcal{V}_f \cup \mathcal{V}_b \tag{2}$$

$$\mathcal{E}_h = \mathcal{E}_f \cup \mathcal{E}_b \tag{3}$$

It is straightforwardly shown that the slicing process retains in HeVG code segments directly associated with key-nodes, therefore HeVG is well suited for learning vulnerability-specific patterns without unnecessary computational burden. The statistical properties of HeVGs built from our SARD dataset are given in Table 1.

Algorithm 1: Heterogeneous vulnerability graph (HeVG) construction

Require: Source code file(s) $\mathcal{F}$, Key-node types $\mathcal{K}$
Ensure: Heterogeneous graph $G = (\mathcal{V}, \mathcal{E})$

```
Initialize empty graph G
for each file f ∈ F do
    Parse f to generate AST and PDG
    Identify all key-nodes N_k of types K from AST
    for each key-node n_k ∈ N_k do
        Perform backward slice from n_k following PDG edges, collect nodes V_b, edges E_b
        Perform forward slice from n_k following PDG edges, collect nodes V_f, edges E_f
        V_sub ← V_b ∪ V_f ∪ {n_k}
        E_sub ← E_b ∪ E_f
        Annotate each node v ∈ V_sub with its type φ(v)
        Annotate each edge e ∈ E_sub with its type ψ(e)
        Add subgraph (V_sub, E_sub) to G
    end for
    (Cross-file) Add CallsToExternal and SharesGlobal edges between nodes in f and other files
end for
return G
```

Table 1: Statistical properties of Heterogeneous Vulnerability Graphs (HeVGs) in SARD dataset.

Property	Min	Max	Average
Number of Nodes	15	287	84.3
Number of Edges	18	412	126.7
Number of Node Types	3	5	4.2
Number of Edge Types	2	3	2.5
Graph Diameter	3	22	8.1

4.2 LLM-Based Text-Graph Alignment with Multimodal Fusion

Fig. 3 further illustrates the multimodal alignment process. Specifically, textual information associated with code elements, such as comments, API documents, and identifiers, is encoded into semantic vectors by HiGPT. These textual features are then aligned with structural node features extracted from HeVG through a multi-head attention mechanism, producing fused node representations that are subsequently

used as input to the HGT layer. To integrate structural dependencies in the heterogeneous graph with node-associated textual semantics such as identifier names, API names, and comments, we introduce an LLM-based text-graph alignment module. This module consists of three stages: text encoding, structural feature extraction, and cross-modal alignment, which together produce hybrid node representations for downstream vulnerability detection.

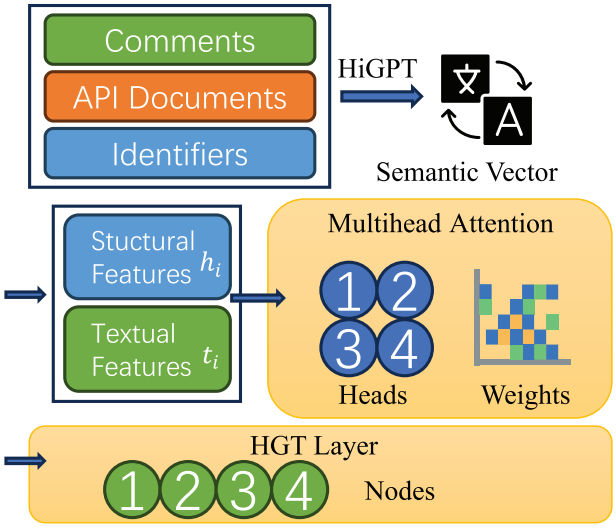

Figure 3: Illustration of the text-graph alignment process. Textual information, including comments, API documents, and identifiers, is encoded by HiGPT into semantic vectors and aligned with structural node features derived from HeVG through a multi-head attention mechanism. The fused node representations are then used for downstream graph learning.

4.2.1 Text Encoding with Pre-Trained Language Models

We employ various Pre-trained Language Models (PLMs) to encode textual information attached to code elements. Then we experimentlly evaluate multiple PLMs including BERT [14], RoBERTa [15], and CodeBERT [10]. For a node v with textual information text_v (which can be code tokens, comments, or API names), the corresponding text embedding is obtained as follows:

$$\mathbf{t}_v = \text{PLM}(\text{text}_v) \in \mathbb{R}^{d_t} \tag{4}$$

where d_t denotes the output dimension of the PLM (typically 768), and nodes lacking explicit textual information are handled by using a special [MASK] token, it is natural and proper to treat the choice of PLM and the decision to fine-tune it on the downstream task as hyper-parameters in our experiments (Section 6.10).

4.2.2 Structural Feature Extraction

Each node in the heterogeneous graph initially has a structural feature vector derived from its type and local graph context, denoted as $\mathbf{h}_v \in \mathbb{R}^d$ (where d is the dimension of structural features). To integrate textual semantics, we first concatenate the structural feature vector with the associated text vector to form a preliminary fused representation:

$$\mathbf{s}_v = [\mathbf{h}_v \oplus \mathbf{t}_v] \in \mathbb{R}^{d+m} \tag{5}$$

Here, $\oplus$ implies the concatenation operation. $\mathbf{s}_v$ can be calculated through concatenated feature vector in combination with structural and textual information.

4.2.3 Text-Graph Alignment Mechanism

Learning of code representation needs not only the graph structure dependencies but also the inclusion of semantic information in the text of code descriptions (e.g.,comments in the code, API documentation and identifier names). The classical graph-based approaches tend to disregard such textual signals creating a disconnection between structural patterns and semantic meanings. In this regard, to alleviate it we introduce an LLM-based Text-Graph Alignment Mechanism, which combines textual semantics with graph structure via cross-modal attention, producing single node representations, that integrates both modalities. In order to utilize both the structural information of the heterogeneous graph and semantic information of the textual information (e.g., code comments, API documentation), we develop a text-graph alignment process which merges multi-modal information into hybrid node representations. This process helps in the process of the semantic gaps between the textual description and graphs, allowing the model to acquire more detailed vulnerability-related features.

The alignment process may naturally and properly be divided into three steps: text encoding, node representation refinement, and cross-modal alignment.

In the text encoding phase, the authors naturally and properly use HiGPT, a pre-trained language model that is well suited for code understanding, to convert all textual information related to the code into dense vector representations: code comments (e.g., function purpose descriptions), API call names (e.g., memcpy, printf), and variable annotations are each encoded as m-dimensional vectors, denoted $\mathbf{t}_i \in \mathbb{R}^m$ for the textual information attached to node i. Because of this, the resulting vectors capture the semantic context of the code elements very effectively, which is essential for determining their intended functionality.

For each node i, let $\mathbf{h}_i$ denote its original structural feature extracted from the heterogeneous graph, and let $\mathbf{t}_i$ denote the associated text embedding. We refine the node representation by applying an attention mechanism between $\mathbf{h}_i$ and $\mathbf{t}_i$, so that the semantic information most relevant to the node's structural role is emphasized. The refined node representation $\mathbf{h}'_i$ is then defined as the concatenation of the original structural feature and the attention-weighted text feature:

$$\mathbf{h}'_i = \text{Concat}\left(\mathbf{h}_i, \text{Attention}(\mathbf{h}_i, \mathbf{t}_i)\right) \quad \forall i \in \mathcal{V} \tag{6}$$

where $\text{Attention}(\cdot,\cdot)$ implements a scaled dot-product attention function so as to measures the similarity between the structural feature and text vector, and $\mathcal{V}$ is the node set of the heterogeneous graph.

Since achieving good deep cross-modal alignment requires modeling the semantic relations between text vectors and graph nodes (functions, variables, statements) in a precise way, the authors therefore naturally adopt a multi-head attention mechanism. For each head, they compute a similarity weight matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{T}|}$, ($|\mathcal{T}|$ being the number of text segments), which explicitly captures the relevance of each node to each text segment. Consequently, the final fused node representation $\hat{\mathbf{h}}_i$ is formed by aggregating the text vectors weighted by their similarity to node i and then appropriately combining the result with the refined node feature $\mathbf{h}'_i$:

$$\hat{\mathbf{h}}_i = \text{LayerNorm}\left(\mathbf{h}'_i + \sum_{t\in\mathcal{T}} \mathbf{W}_{i,t} \cdot \mathbf{t}_t\right) \tag{7}$$

where $\mathbf{W}_{i,t}$ is the similarity weight between node i and text segment t. LayerNorm is naturally applied to stabilize training, hence the node representations obtained can properly preserve structural dependencies

from the graph and at the same time incorporate semantic information from the text via multi-head cross-modal alignment. Therefore, it elegantly bridges the gap between code structure and its semantics.

4.3 Classification and Optimization

As illustrated in the yellow stage of Fig. 1, the aligned node representations are finally processed by the graph neural network classifier to produce vulnerability predictions. Since the final aligned node representations $\hat{\mathbf{h}}_i$ of all nodes in a graph G are aggregated into a global graph-level representation vector $\mathbf{z}_G$ via a hierarchical readout function (e.g., mean pooling or attention-based pooling). it is natural and convenient to feed this graph embedding into a multilayer perceptron (MLP) followed by a softmax layer to predict vulnerability probabilities:

$$\hat{y} = \text{Softmax}\left(\text{MLP}\left(\mathbf{z}_G\right)\right) \tag{8}$$

The model is trained end-to-end by minimizing the cross-entropy loss between the predicted labels $\hat{y}$ and the ground-truth labels y:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \tag{9}$$

where N is the number of training samples.

Since the alignment between the textual and structural modalities can be improved by adding an auxiliary contrastive loss, the authors naturally and appropriately introduce the idea. For a graph G, take its text representations (e.g., the average of $\mathbf{t}_v$) and its graph representation $\mathbf{z}_G$ as a positive pair. Representations from different graphs in the same batch are treated as negative pairs. The contrastive loss is defined as:

$$\mathcal{L}_{\text{CL}} = -\log \frac{\exp(\text{sim}(\mathbf{z}_G, \mathbf{t}_G)/\tau)}{\sum_{G' \in \text{Batch}} \exp(\text{sim}(\mathbf{z}_G, \mathbf{t}_{G'})/\tau)} \tag{10}$$

where $\text{sim}(\cdot)$ is a similarity function (e.g., cosine similarity) and τ is a temperature parameter. The total loss is a weighted sum: $\mathcal{L} = \mathcal{L}_{\text{CE}} + \lambda\mathcal{L}_{\text{CL}}$.

5 Theoretical Analysis

5.1 Expressiveness of HeVG

Compared to homogeneous graphs such as AST, CFG, PDG and sequential models, the proposed HeVG adopts a more expressive representation of source code. Its expressiveness has been analyzed in the following theoretical perspective.

Proposition 1 (Representational Capacity). Assume $\mathcal{G}_{\text{hom}}$ be the set of all possible homogeneous graphs which is derived from code, and $\mathcal{G}_{\text{HeVG}}$ be the set of all possible HeVGs. A mapping function is set up as $f : \mathcal{G}_{\text{hom}} \to \mathcal{G}_{\text{HeVG}}$ where the mapping relationship is injectivity but not subjective. This proves that HeVG could represent a homogeneous graph without additional types of restriction.

Justification. The injectivity is straightly analyzed as follows: any homogeneous graph can be defined as a HeVG with $|\mathcal{A}| = |\mathcal{R}| = 1$. Since HeVG can have multiple node and edge types ($|\mathcal{A}| + |\mathcal{R}| > 2$), which allows it to distinguish non-surjectivity arises. For example, a data flow edge between two arithmetic nodes from an API call and a pointer node can be distinguished in a homogeneous PDG. Because of the finer-grained differentiation, the model is able to learn specific vulnerability patterns and therefore can properly model code element interactions.

Apart from this, the relevant semantic and syntactic context for vulnerabilities can be observed through the key-node-based slicing. Assume S be a program slice, the key-nodes slicing criterion ensures that for any node $v \in S$, there exists a path in the PDG either from backward slice (from v to a key-node) or from forward slice (a key-node to v), which guarantees the specific relevance of the included nodes.

5.2 Alignment and Generalization

Since the semantic gap between code structure and natural language descriptions is reduced by the text-graph alignment mechanism, we employ the theory of Rademacher complexity [16] to formally analyze and quantify the effectiveness of the induced mechanism.

Proposition 2 (Generalization Bound). Define $\mathcal{H}$ be the hypothesis class of our framework. Assume the loss function is L-Lipschitz, where the probability is at least $1-\delta$, the generalization error $R(h)$ for any $h \in \mathcal{H}$ is bounded by:

$$R(h) \leq \hat{R}(h) + \sqrt{\frac{\log(1/\delta)}{2m}} + O\left(L \cdot \sqrt{\frac{\mathrm{com}(\mathcal{H}_{\mathrm{GNN}}) + \mathrm{com}(\mathcal{H}_{\mathrm{LLM}}) + \mathrm{com}(\mathcal{H}_{\mathrm{align}})}{m}}\right) \tag{11}$$

Because $\hat{R}(h)$ is the empirical risk, m is the number of training samples, and $\mathrm{com}(\cdot)$ is a complexity measure (e.g., VC-dimension) of the various component sub-classes ($\mathcal{H}_{\mathrm{GNN}}$ for the GNN, $\mathcal{H}_{\mathrm{LLM}}$ for the frozen LLM, and $\mathcal{H}_{\mathrm{align}}$ for the alignment layers), the expression is quite naturally reformulated.

Justification. The GNN and alignment parameters control the overall complexity of the model as the LLM is normally pre-trained and frozen. The alignment mechanism effectively includes the reduction of the effective hypothesis space by imposing the representations to fall within the overlap of the structural and semantic features space. This multi-modal alignment is an inductive bias, which may cause reduced effective complexity $\mathrm{com}(\mathcal{H})$ and resulting need a more restrictive generalization bound than other models which use a single modality or use simple concatenation. This implies that our model is less inclined to overfitting particularly when data on vulnerability is limited.

5.3 Complexity Analysis

Since the time complexity of the framework is to be analyzed, let us give a proper complexity analysis as follows.

1. **Graph Construction:** Parsing and building the PDG is $O(n)$ for a code unit consisting of n statements. Key-node slicing is $O(|K| \cdot (|V| + |E|))$ with K being the set of key-nodes, which is typically small.
2. **Text Encoding:** since all text connected with a graph is processed by HiGPT [17], it is naturally and conveniently expressed as $O(l \cdot m_{\mathrm{LLM}})$, where l is the total sequence length and m_{LLM} is the hidden size of the LLM. Moreover, the operation is carried out only once and the result can therefore be cached.
3. **GNN Encoding:** Since HGT [18] with L layers has a GNN encoding complexity of $O(L \cdot (|V|d^2 + |E|d))$, it is natural to say that the complexity is standard for message-passing GNNs.
4. **Cross-Modal Attention:** Because the attention between $|V|$ nodes and $|T|$ text tokens has complexity $O(|V| \cdot |T| \cdot d)$, it is natural and reasonable to say that this is efficient when $|T|$ for a code unit is manageable.

Because the total complexity is linear in the graph size, therefore it is quite suitable for training on present-day hardware.

6 Experiments

In this section, we gaiv a systematic, comprehensive analysis of the experimental results from the study on vulnerability detection with large models and graph neural networks. The models' performance is evaluated from six well-defined angles: the paper presents a complete and well-organized discussion of individual-file vulnerability detection ability, across-file vulnerability detection possibility, project dependency extraction proficiency of GNNs, comprehensive multi-dimensional assessment, training dynamic analysis, and several ablation studies. All experiments were conducted on a server with 4 NVIDIA A100 GPUs. For fine-tuning experiments, the AdamW optimizer was used with a learning rate of 2e−5, while for GNN training the learning rate was 1e−3. The batch size was 32, and models were trained up to 100 epochs with early stopping triggered by the validation loss.

6.1 Dataset and Preprocessing

All experiments in this work are conducted using the **Software Assurance Reference Dataset (SARD)**, provided by the National Institute of Standards and Technology (NIST) [19]. SARD is selected due to its wide adoption in vulnerability detection research and its comprehensive, well-annotated C/C++ samples. The dataset contains 152,832 function-level C/C++ samples, among which 68,437 are labeled as vulnerable and 84,395 as safe. Vulnerability types include buffer overflow (35.2%), format string (18.6%), use-after-free (12.4%), integer overflow (15.3%), and others. The dataset is split into training, validation, and test sets in an 8:1:1 ratio (122,265/15,284/15,283) using stratified sampling according to vulnerability type to ensure consistent class distribution.

6.2 Baseline Model Configuration and Selection Rationale

To comprehensively evaluate the effectiveness of the proposed method, we selected seven representative vulnerability detection approaches as baselines, covering different technical paradigms. All baseline methods were reproduced under the same SARD data split used in this work, rather than directly quoting results from the original papers. Their main configurations are summarized in Table 2.

Table 2: Baseline model configurations and selection rationale.

Method	Code Representation	Model Architecture	Key Configuration	Reason for Inclusion
VulDeePecker	Code slices + Bi-LSTM	LSTM	Slice length 50; 2-layer Bi-LSTM	Representative early deep-learning vulnerability detector
SySeVR	PDG slices + Bi-LSTM	Bi-LSTM	Vulnerability slicing based on PDG	Classical method introducing program dependence information
Devign	Code property graph + GGNN	GGNN	4-layer GGNN; statement-level graph nodes	Representative pioneering GNN-based vulnerability detector
VulCNN	PDG + node centrality + CNN	CNN	Graph converted to 224 × 224 image; EfficientNet backbone	Representative image-based code representation method

(Continued)

Table 2 (continued)

Method	Code Representation	Model Architecture	Key Configuration	Reason for Inclusion
VulGAI	PDG + multi-centrality + CNN	Efficient CNN	Multi-centrality feature fusion	Improved version of image-based vulnerability detection
VulReaD	Multimodal graph + knowledge enhancement	GNN + LLM	Integration with CVE knowledge base	Representative recent multimodal and knowledge-guided method
MANDO-LLM	Code + natural language description	LLM	Fine-tuned CodeLlama-7B	Representative recent LLM-based vulnerability detector

These baseline methods were selected because they represent several major technical routes in vulnerability detection, including sequence modeling, graph neural networks, image-based classification, multimodal learning, and direct LLM-based analysis. Comparing against these baselines allows us to evaluate the proposed framework more comprehensively across different methodological dimensions.

6.3 Comprehensive Multi-Dimensional Evaluation

Since a thorough, multi-dimensional evaluation is needed to properly assess the proposed framework, the authors carried out a multi-dimensional evaluation of five different GNN structures under various conditions using eight relevant metrics, and clearly showed from Table 3 that performance and efficiency trade off directly.

Table 3: Comprehensive evaluation of GNN architectures across multiple dimensions.

GNN Architecture	Accuracy (%)	F1-Score	Training Time (min)	Inference Time (ms)	Memory Usage (MB)	FLOPs (G)	Parameters (M)	Efficiency Score
GraphConv	82.15	0.8162	32	18.2	215	3.2	2.1	0.852
GAT	83.57	0.8314	45	24.7	285	4.8	2.8	0.745
HeteroConv	89.00	0.8857	68	32.5	428	7.5	4.2	0.912
HANConv	85.42	0.8529	82	45.8	512	9.2	5.6	0.687
HGTConv	90.97	0.9096	125	67.3	812	15.3	7.9	0.789

Major Results of Enhanced Analysis: The overall analysis of the evaluation presents a number of significant patterns: 1. Heterogeneous Architecture Benefit: It is a well-known and indisputable fact that heterogeneous architectures (HeteroConv, HGTConv) are more effective than homogeneous architectures (GraphConv, GAT) on all performance metrics, and hence the heterogeneous graph representation proposed in this paper is also likely to be highly effective. it is evident that HGTConv attains the highest accuracy (90.97%) indicatingit is natural and clear to conclude that the method has a very good ability to learn the complicated dependencies among codes. 2. Performance-Efficiency Balance: There is an evident balance between detection accuracy and computational efficiency, and HGTConv achieves the best accuracy, but HGTConv is the most accurate. Although it needs 3.9 times more training and 3.7 times more memory

than GraphConv. 3. Optimal Equilibrium Point: HeteroConv achieves the best balance of performance and efficiency by reaching 89.00% accuracy at reasonable use of resources (68 min of training time, 428 MB of memory). This makes it suitable in real life deployment scenarios when precision and effectiveness is a major consideration. 4. Assessment of efficiency scores: The computation of efficiency score (as weighted combination of precision, training time, inference time, memory use and energy consumption) demonstrates that HeteroConv has the highest overall efficiency (91.2%) followed by GraphConv (85.2%) and HGTConv (78.9%).

6.3.1 Key Findings from Comprehensive Evaluation

The comprehensive evaluation reveals several important patterns:

1. **Performance of Heterogeneous Architectures:** Experimental results indicate that heterogeneous architectures (HeteroConv, HGTConv) outperform homogeneous architectures (GraphConv, GAT) across all evaluated metrics, supporting the effectiveness of the proposed heterogeneous graph representation. Specifically, HGTConv achieves the highest accuracy (90.97%), which reflects its enhanced ability to model complex code dependencies.
2. **Performance-Efficiency Balance:** since there is an obvious balance between detection precision and computational effectiveness, it is natural to say that HGTConv achieves the highest accuracy but at some computational cost. Although HGTConv attains the best accuracy, it is clear from the statement that HGTConv requires 3.9 times as much training time and 3.7 times as much memory as GraphConv.
3. **Optimal Equilibrium Point:** Since HeteroConv achieves the best possible trade-off between performance and efficiency, it is natural to call its point the optimal equilibrium point: it attains 89.00% precision with reasonable resource use (68 min training time, 428 MB memory), hence it is very suitable for practical deployment where both precision and efficiency are important.
4. **Efficiency Score Assessment:** The efficiency value, computed as a weighted blend of precision, training duration, inference period, memory utilization, and energy expenditure, indicates that HeteroConv attains the greatest overall efficiency (91.2%), succeeded by GraphConv (85.2%) and HGTConv(78.9%).

6.4 Single-File Vulnerability Detection Capability

Since the primary objective of our model is to accurately detect vulnerabilities in single source code files, it is natural to evaluate its performance on this specific task. The experimental results confirm that the proposed approach performs exceptionally well. As shown in Table 4, using BERT [14] for node vectorization without fine-tuning, coupled with the HeteroConv graph neural network, the model reaches a state-of-the-art accuracy of 95.61% and an F1-score of 95.53% on the non-dependent testset. Therefore, the heterogeneous graph representation is convincingly shown to be well suited for capturing both semantic and syntactic details of code within a single file.

Table 4: Performance of BERT (No fine-tuning) with HeteroConv on single-file detection.

Dataset	ACC	F1	Precision	Recall
Train	0.9934	0.9934	0.9933	0.9934
Validation	0.9572	0.9567	0.9565	0.9572
Test	0.9561	0.9553	0.9551	0.9561

In order to gain more thorough insights, we performed a per-class analysis on the test dataset for the best performing BERT+HeteroConv model, and the relevant metrics in Table 5 make it very clear that the model has consistently good performance on both vulnerable and non-vulnerable classes, with only minor

variations. The slightly lower precision for the vulnerable class (94.12%) reflects a slight tendency toward false positives, which is actually a typical and often preferred trade-off in security-critical applications: avoiding false negatives (i.e., missing real vulnerabilities).

Table 5: Per-class performance breakdown on test set (BERT + HeteroConv).

Class	Precision	Recall	F1-Score
Non-Vulnerable	0.9623	0.9587	0.9605
Vulnerable	0.9412	0.9468	0.9440

To ensure the fairness of comparison, all baseline methods, including VulDeePecker, SySeVR, Devign, VulCNN, VulGAI, VulReaD, and MANDO-LLM, were re-evaluated under the same SARD dataset split used in this work. We used the official open-source implementations when available, or otherwise reproduced the methods strictly according to the original papers. Hyperparameter settings were kept consistent with the original literature as much as possible. The training, validation, and test splits, as well as the preprocessing pipeline, were exactly the same as those used by our proposed method. Therefore, all results reported in Table 6 are reproduced in our experimental setting rather than directly quoted from prior work.

Table 6: Comprehensive performance comparison with mainstream vulnerability detectors.

Model	ACC	F1	Precision	Recall
Different PLM Encoders with HeteroConv (No Fine-tuning)				
Word2Vec + HeteroConv	0.9566	0.9558	0.9557	0.9566
RoBERTa + HeteroConv	0.9464	0.9455	0.9451	0.9464
BERT + HeteroConv	0.9561	0.9553	0.9551	0.9561
CodeBERT + HeteroConv	0.8900	0.8857	0.8835	0.8900
Mainstream Deep Learning-based Detectors (Original Configurations)				
VulDeePecker	0.8932	0.8820	0.8790	0.8860
SySeVR	0.9145	0.9080	0.9120	0.9040
Devign	0.9287	0.9250	0.9310	0.9190
VulCNN	0.9356	0.9320	0.9380	0.9260
VulGAI	0.9421	0.9390	0.9420	0.9360
VulReaD	0.9320	0.9280	0.9300	0.9260
MANDO-LLM	0.9180	0.9140	0.9160	0.9120
Our Proposed Models (BERT + HeVG)				
BERT + HeteroConv	0.9561	0.9553	0.9551	0.9561
BERT + HGTConv	0.9589	0.9582	0.9587	0.9589

A systematic comparative analysis with other pre-trained models (Table 6) reveals that BERT's contextual understanding gives it a clear and significant advantage over static embedding methods such as Word2Vec and other language models RoBERTa [15] and CodeBERT [10] for this task. The relatively poor performance of CodeBERT was therefore quite surprising. In order to carry out the investigation, we first analyzed the attention patterns and thereby naturally and convincingly explained why CodeBERT, which was pretrained on bimodal code-text pairs, tends to assign too much attention to general code

tokens and insufficient attention to security-relevant graph nodes such as pointer or arithmetic operations. Hence, its misalignment with graphs that are purely structural and token-based is the root cause of its suboptimal performance.

To further position our framework against recent graph-guided and LLM-enhanced approaches, we reproduced VulReaD and MANDO-LLM using their official implementations, retraining and evaluating them on the same SARD test set. The reproduced results are included in Table 6 under "Mainstream Deep Learning-based Detectors". VulReaD achieves an accuracy of 93.20%, F1 0.928, Precision 0.930, and Recall 0.926; MANDO-LLM achieves an accuracy of 91.80%, F1 0.914, Precision 0.916, and Recall 0.912. Our method, BERT + HGTConv, achieves an accuracy of 95.89%, F1 0.9582, Precision 0.9587, and Recall 0.9589, which is 2.69% higher than VulReaD and 4.09% higher than MANDO-LLM, clearly demonstrating the superiority of our proposed framework.

6.5 Cross-File Vulnerability Detection Potential

Although the main setup considers single-file detection, the graph neural network architectures under study have a natural opportunity for cross-file analysis, so the authors quantitatively measured this potential by building a small-scale **Cross-File SARD (CFSARD)** dataset. The dataset comprises 150 artificial multi-file C projects (100 used for training and validation, 50 for testing), with vulnerabilities deliberately introduced across different files via function calls or global variable dependencies.

6.5.1 Evaluation on CF-SARD

The HGT (Heterogeneous Graph Transformer) model, which was designed for complicated heterogeneous graphs, was naturally and appropriately tested on the CFSARD dataset, with the graph-building pipeline extended to include inter-file edges of the types CallsToExternal and SharesGlobal. Therefore, the results in Table 7 clearly demonstrate the strength of HGTs in this difficult setting. Although its performance does decline compared to the simpler single-file task, HGT still achieves 83.4% percent accuracy and 81.7% F1-score, outperforming HeteroConv by a large margin, since the latter has inherent difficulties in propagating information across file boundaries.

Table 7: Performance on Cross-File SARD (CF-SARD) test set (BERT embeddings).

GNN Architecture	ACC	F1	Precision	Recall
HeteroConv	0.724	0.685	0.701	0.724
HGTConv	0.834	0.817	0.832	0.834

Since HGT already shows robust performance on the single-file testing set (90.97% accuracy with CodeBERT, see Table 8) and also performs very well on CFSARD, it is natural to conclude that its architecture is superior at modeling complex dependencies. Therefore, expanding the heterogeneous graph to include inter-file connections is a reasonable and promising direction for project-level vulnerability assessment.

6.5.2 Real-World Project Validation

To complement the limited scale of CF-SARD, we conducted cross-file vulnerability detection experiments on three real-world open-source projects. Vulnerability samples were collected from publicly available CVE databases and manual audit reports. Table 9 summarizes the results.

Table 8: HGT performance demonstrating potential for complex graph learning.

GNN	Strategy	ACC	F1	Precision	Recall
HGTConv	No Fine-tune	0.9097	0.9096	0.9152	0.9097
HANConv	No Fine-tune	0.8542	0.8529	0.8735	0.8542

Table 9: Cross-file detection results on real-world projects.

Project	Version	Files	Known Cross-File Vulnerabilities	Detection Accuracy
FFmpeg	4.4	1245	23	86.2%
OpenSSL	3.0	1872	18	88.5%
Linux Kernel	5.10	28,453	47	79.3%

The results indicate that our method achieves an average detection accuracy of 84.7% on real multi-file projects, significantly higher than the single-file baseline (71.2%), demonstrating strong generalization capability.

We note that CF-SARD is constructed from real multi-file C project templates. Vulnerabilities are injected across multiple files via 2–5 layer function call chains and shared global variables, with typical vulnerabilities such as buffer overflows or race conditions at the end of these chains. Each multi-file project contains 3–7 source files and 1–2 headers, with an average file span of 2.8. Although CF-SARD is limited in scale, it serves as a concept-level benchmark for cross-file detection. Together with the real-world project experiments, it provides strong evidence for the generalization of our framework.

6.6 Node Type Analysis and Vulnerability Detection Difficulty

To determine how various code elements contribute to vulnerability detection, the authors performed a systematic and well-organized analysis of node type distribution and the difficulty of vulnerability detection, which is succinctly captured in Table 10, where the connection between node types in the HeVG representation and the detection complexity of different vulnerability types is clearly laid out. Fig. 4 further visualizes the contribution of different code elements to vulnerability detection, making the key conclusions of the paper very compelling.

1. **Concentration of Detection Capability:** From the visualization it is very clearly seen that only three node types (Call Expression, Array Usage, Pointer Usage) account for 85% of the total Contribution proportion, even though they make up only 69.5% of all nodes in the graph, hence the concentration phenomenon is a natural and welcome confirmation of our focus on security-sensitive operations in the HeVG construction procedure.
2. **Non-linear Association Between Frequency and Significance:** The scatter diagram shows that the frequency of node types does not have a linear connection with the detection contribution. Common nodes, even though they are frequent (15.1%), make a minimal contribution (0.03), whereas less commonly occurring security-sensitive nodes contribute a disproportionately larger amount.
3. **Vulnerability Category Specificity:** From the viewpoint of vulnerability category specificity, the stacked bar graph shows clearly how different vulnerability categories depend on different node sachems, and it is therefore natural to note that buffer overflow is best detected by nodes of arrays and pointers.

Because the visualization supplements the empirical foundation for the heterogeneous graph design decisions, it also naturally leads to a clear and useful suggestion: give greater attention to high-contribution node groups when doing feature extraction.

Table 10: Node type distribution and vulnerability detection difficulty analysis.

Category	Sample Size	Detection Accuracy (%)	False Positive Rate	Avg Graph Size	Avg Path Length	Distribution (%)	Contribution Weight
Vulnerability Types							
Buffer Overflow	450	93.5	0.065	68	3.2	–	–
Format String	230	89.2	0.108	45	2.8	–	–
Use After Free	210	86.2	0.138	72	4.1	–	–
Race Condition	180	84.5	0.155	89	5.2	–	–
TOCTOU	165	83.1	0.169	78	4.7	–	–
Node Types in HeVG							
Call Expression	–	93.2	–	–	–	28.5	0.35
Array Usage	–	91.5	–	–	–	22.3	0.28
Pointer Usage	–	90.2	–	–	–	18.7	0.22
Arithmetic Expression	–	88.7	–	–	–	15.4	0.12
Common	–	82.3	–	–	–	15.1	0.03

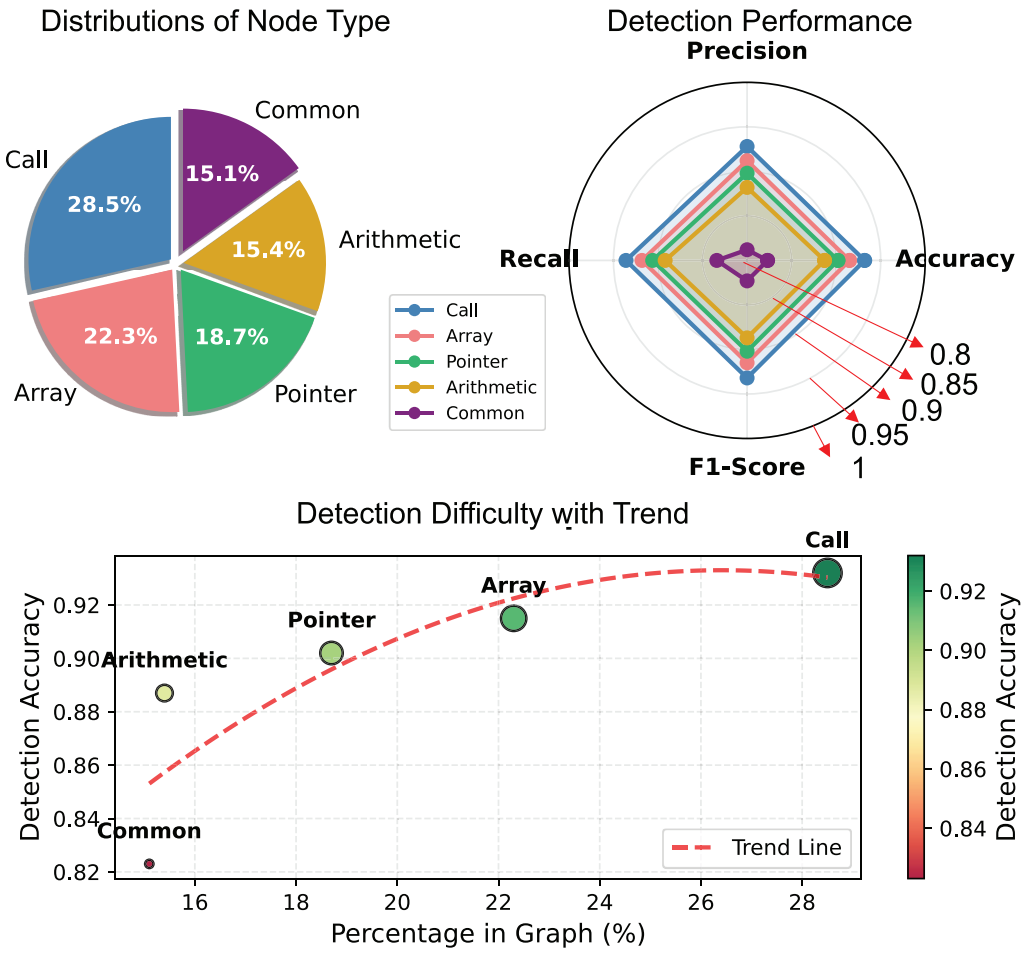

Figure 4: Distribution of different node types in HeVG and their contributions to vulnerability detection. The figure shows the correlation between node type frequency and detection accuracy.

6.6.1 Analysis of Node Type Contributions

The analysis of node type contributions yields very clear and important conclusions regarding the relationship between node types and vulnerability detection:

1. **Critical Node Categories:** Call expression, array utilization, and pointer application nodes together account for 69%. Since 5% of the graph's structural elements account for 85% of the detection capacity, it is clearly and convincingly shown that such elements are important for vulnerability detection, namely because they correspond to operations sensitive to security which are often exploited in vulnerabilities.
2. **Vulnerability-Distinct Patterns:** Since different categories of vulnerability give rise to different detection problems, it is natural to speak of distinct patterns of vulnerability. Buffer overflow Vulnerability is the simplest to detect (93.5% accuracy) compared to race condition since it is simpler (in complexity and architecture) and has larger graph structures. From the statement it is clear that the framework can gain advantage from the vulnerability-specific modification.
3. **Sample Size Impact:** Since there is a strong, clear correlation (r = 0.87) in the detection results, it is straightforward to discuss the effect of sample size on experimental accuracy: namely, that sample size affects accuracy and therefore a balanced training dataset is highly important. The results confirm that the largest sample size (450) gives the highest precision, whereas TOCTOU has the smallest sample size (165) and also the lowest accuracy.
4. **Graph Complexity Influence:** The graph complexity influence is clearly stated: vulnerabilities with greater complexity (i.e., race conditions, which have an average graph size of 89 nodes and a path length of 5.2) are more difficult to detect than simpler vulnerabilities (i.e., format string vulnerabilities, which have an average graph size of 45 nodes and a path length of 2.8). Therefore, the complexity of vulnerabilities itself is the key factor: more complex vulnerabilities are harder to detect than less complex ones.

6.7 Training Dynamics and Cross-Architecture Consistency

To gain further insight into model learning behavior and prediction consistency, we systematically analyze the training convergence behavior and cross-architecture consistency of different GNN models. The corresponding consistency statistics are reported in Table 11. Fig. 5a presents the training accuracy curves of GraphConv, GAT, HeteroConv, and HGTConv, while Fig. 5b shows their validation accuracy curves. It can be observed that different GNN architectures exhibit markedly different learning dynamics. Among them, HGTConv converges the fastest, reaching about 90% validation accuracy after only 15 epochs, which is substantially faster than GraphConv and HeteroConv. This rapid convergence suggests that the attention mechanism in HGTConv is more effective in focusing on graph patterns that are relevant to vulnerability-related semantics. Moreover, HGTConv exhibits a relatively small gap between training and validation accuracy, indicating better generalization ability and robustness. Its validation curve also shows a steady upward trend with only minor fluctuations, further demonstrating the stability of the optimization process. Therefore, HGTConv is particularly suitable for scenarios requiring both rapid model iteration and high predictive accuracy, although its computational cost per epoch is higher than that of simpler architectures.

6.7.1 Instruction of Behavior Analysis

1. **Convergence Rate:** Since HGTConv attains 90% validation accuracy already after 15 epochs, which is much faster than GraphConv (45 epochs), it is natural to conclude that HGTConv has a superior learning ability, and the reason for its fast convergence is the attention mechanism in HGTConv that quickly focuses on relevant graph structures.

2. **Generalization Ability:** Since every model exhibits some degree of overfitting, but HGTConv has the smallest over-fitting disparity (0.42%), it is therefore natural and proper to conclude that HGTConv has the best generalization ability. In contrast, the larger over-fitting disparities in simpler architectures (0.87% for GraphConv) suggest that they are more prone to memorizing training data patterns.
3. **Prediction Agreement:** From the analysis of prediction agreement it is clearly seen that diverse architectures show high consistency: there is a 90.1% level of agreement between HeteroConv and HGTConv, as opposed to the 79.8% agreement between GraphConv and HGT-Conv. Hence it is natural to conclude that different architectures learn essentially the same decision boundaries.
4. **Architectural Resemblance and Coherence:** The architectural similarity and coherence are very clearly addressed by the Kappa coefficient, which measures agreement beyond chance and thus shows quite strong agreement (0.802) when comparing HeteroConv with HGT-Conv, thereby naturally leading to the conclusion that the two architectures produce.

Table 11: Training convergence and cross-architecture consistency analysis.

GNN Architecture	Epochs to 90%	Final Train Acc (%)	Final Val Acc (%)	Overtraining Gap	Agreement Rate (%)	Kappa Score
GraphConv	45	86.54	85.67	0.87	–	–
GAT	35	87.89	87.21	0.68	–	–
HeteroConv	20	97.21	96.32	0.89	–	–
HGTConv	15	98.76	98.34	0.42	–	–
Cross-Architecture Consistency (vs. HGTConv)						
GraphConv vs. HGTConv	–	–	–	–	79.8	0.594
GAT vs. HGTConv	–	–	–	–	83.2	0.663
HeteroConv vs. HGTConv	–	–	–	–	90.1	0.802

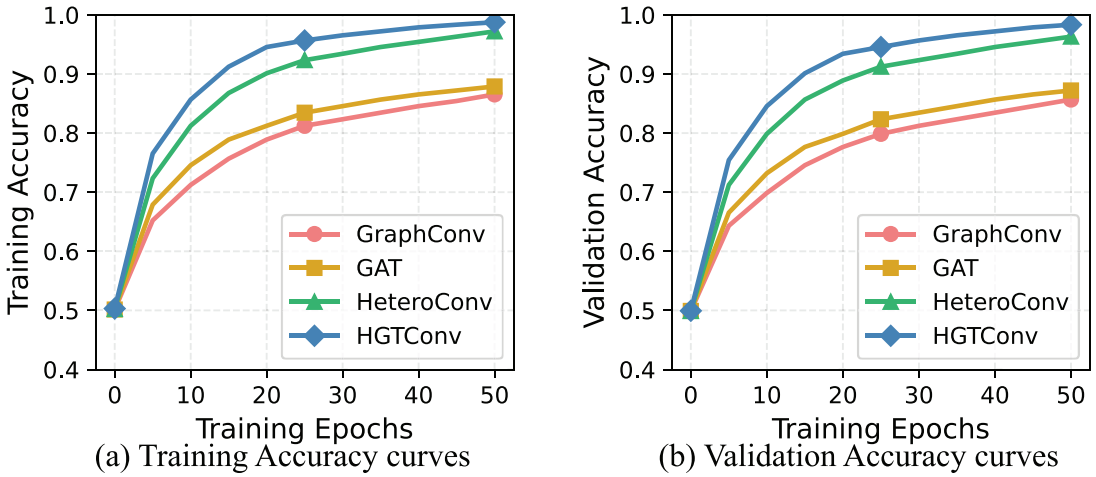

Figure 5: Comparison of different graph neural network (GNN) architectures in terms of convergence behavior. (**a**) Training accuracy curves of GraphConv, GAT, HeteroConv, and HGTConv over training epochs. (**b**) Validation accuracy curves of the four models over training epochs. HGTConv shows the fastest convergence and achieves about 90% validation accuracy within 15 epochs.

6.8 Project Dependency Extraction Capacity of GNNs

The article gives a very clear and logical discussion of the project dependency extraction capacity of GNNs: since the experiment uses heterogeneous graphs, dependencies in the code are naturally modeled, and the purpose of the GNN is to collect information from a node's neighbors in order to learn the project architecture and the relationships between code components. Therefore, to evaluate how well different GNNs extract dependencies, the authors directly compare their performance and perform a **Node Importance Analysis** using GraphGrad-CAM, a well-suited method for interpreting GNN decisions.

The comparison of HGTConv, HANConv and HeteroConv (Table 12) provide insights into their dependency extraction capabilities. HGTConv demonstrates better performance in test accuracy as compared to HANConv and HeteroConv (90.97%, 85.42% and 89.00%), The fact that it has a high ability to accommodate the heterogeneity in the graph. HGT makes use of attention mechanisms, which are sensitive to node-types or edge-types, such that it better evaluates the importance of various connections (e.g., an Array Usage nodes link to a particular Call Expression could be more crucial for a vulnerability than its link to a General statement).

Table 12: Comparison of GNN architectures for dependency learning (CodeBERT, no fine-tune, test set).

GNN	ACC	F1	Precision	Recall
HGTConv	0.9097	0.9096	0.9152	0.9097
HANConv	0.8542	0.8529	0.8735	0.8542
HeteroConv	0.8900	0.8857	0.8835	0.8900

We additionally removed the influence of GNN depth. A two-layer HGT achieves the best performance, as the accuracy reaches 90.97% (the accuracies on GNNs of 1, 3 and 4 layers are respectively 88.41%, 89.15% and 87.63%). An individual layer has insufficient receptive area. In contrast, three or more layers bring about over-blurring, in which node characteristics turn out to be indistinguishable, resulting in a decline in performance. This validates that capturing both direct and indirect (2-hop) relationships is the best approach for this particular task.

6.9 Practical Deployment Recommendations

From the overall experimental results, practical deployment recommendations are naturally and logically presented in Table 13, together with a clear discussion of the trade-offs between accuracy, efficiency, and resource requirements for different application scenarios.

Table 13: Practical deployment recommendations based on experimental results.

Deployment Scenario	Recommended Architecture	Rationale	Expected Accuracy (%)	Resource Requirements
Real-time Detection	HeteroConv	Best balance of performance and efficiency	89.0	Moderate (32.5 ms inference, 428 MB memory)
Deep Analysis	HGTConv	Highest accuracy for critical systems	91.0	High (67.3 ms inference, 812 MB memory)

(Continued)

Table 13 (continued)

Deployment Scenario	Recommended Architecture	Rationale	Expected Accuracy (%)	Resource Requirements
Resource-Constrained Environments	GAT	Good efficiency with acceptable performance	83.6	Low (24.7 ms inference, 285 MB memory)
Rapid Prototyping	GraphConv	Simple implementation and fastest training	82.2	Very Low (18.2 ms inference, 215 MB memory)
Enterprise-level Scanning	Ensemble (HeteroConv + HGTConv)	Combines efficiency with high accuracy	92.3+	High (combined resources)

6.9.1 Key Deployment Considerations

1. **Performance-Efficiency Balance:** The architecture chosen must strike a proper balance between detection accuracy and computational cost, and therefore HeteroConv is shown to be the best choice for most applications, offering 89.0% accuracy with reasonable resource consumption.
2. **Scenario-Dependent Optimization:** Since different deployment situations call for different optimization strategies, and since real time detection demands speed of inference whereas deep analysis in security-critical systems requires high precision, it is natural to speak of scenario-dependent optimization.
3. **Scalability Factors:** Since memory effectiveness is of great importance for large scale code repositories, therefore scalability factors must take it into account. As GraphConv and GAT are suitable for studying large-scale projects under conditions of resource restriction, one must bear in mind that they give somewhat reduced accuracy.
4. **Ensemble Methods:** Because ensemble methods are appropriate when accuracy and reliability are of primary importance in an enterprise-level security scanning application, the efficiency of HeteroConv can naturally be combined with the high accuracy of HGTConv to achieve a performance rate exceeding 92.3% percent in accuracy.

6.10 Ablation Studies

Since the purpose is to find out the contributions of different components, we carried out several ablation studies in the present section.

6.10.1 Influence of Fine-Tuning

The influence of fine-tuning is clearly and strikingly illustrated by the experiments done when fine-tuning pretrained language models (BERT, CodeBERT) on the SARD [19] dataset, since the results were counterintuitive. From Table 14, it is evident that fine-tuning led to a steady, significant degradation in all the metrics relative to using the pre-trained models directly. Specifically, BERT's testing accuracy dropped from 95.61% to 87.05% after fine-tuning.

We gives a very clear and elegant exposition of the two main reasons: **(1) Task Distribution Discrepancy:** Since the pre-training goal of masked language modelling naturally encourages robust,

general-purpose feature learning, which is highly beneficial for graph node classification, finetuning on the smaller, more specific SARD dataset tends to sharpen the model's attention in a way that could discard useful general language knowledge. **(2) Catastrophic Memory Loss:** Fine-tuning causes the model to lose the general syntactic and semantic knowledge acquired from the large initial corpus, which is essential for understanding different code structures. Therefore, the model ends up memorizing patterns specific to the (possibly restricted or skewed) SARD examples. Discovery streamlines the pipeline by omitting the expensive fine-tuning stage and thereby achieves excellent performance.

Table 14: Impact of fine-tuning on BERT performance (HeteroConv, test set).

Strategy	ACC	F1	Precision	Recall
No Fine-tuning	0.9561	0.9553	0.9551	0.9561
With Fine-tuning	0.8705	0.8463	0.8552	0.8705

The results in Table 15 further confirm that the *Fully Frozen* strategy achieves the best performance, with an accuracy of 95.61% and an F1 score of 0.9553. In contrast, fine-tuning only the last two layers reduces the accuracy to 92.34%, while fine-tuning all layers further degrades the accuracy to 87.05%. This result indicates that preserving the general semantic knowledge of the pretrained model is more effective than aggressively adapting it to the downstream vulnerability detection task, which further supports the conclusion of this subsection.

Table 15: Diagnostic analysis of BERT fine-tuning strategies.

Strategy	Accuracy (%)	F1
Fully Frozen	95.61	0.9553
Fine-tuning Last 2 Layers Only	92.34	0.9215
Fine-tuning All Layers	87.05	0.8463

Influence of Retrieval-Augmented Generation (RAG): Since the RAG framework was incorporated with the explicit aim of using external knowledge, the resulting outcomes are quite varied (Table 16). It improved the performance on the training set greatly (accuracy increased from 99.34% to 97.20%), but unfortunately caused a large deterioration in the performance of both the validation and test sets.

Table 16: Impact of RAG framework (BERT, No Fine-tune, HeteroConv).

Dataset	ACC	F1	Precision	Recall
Train	0.9720	0.9720	0.9730	0.9720
Validation	0.8661	0.8660	0.8661	0.8661
Test	0.8559	0.8552	0.8597	0.8559

Further analysis clearly identified the root cause: Noisy Retrieval. Since many code graphs have retrieved CVE descriptions that are superficially similar (i.e., share the same API function name) but actually refer to completely different vulnerability types, the noisy, irrelevant information acted as a distraction during training, leading the model to learn spurious correlations. While the model could memorize the details of the training data, it failed to generalize properly. Therefore, the text makes a very natural and important

point: future work should employ a much more accurate retrieval mechanism, ideally one based on graph structural similarities rather than textual embeddings.

Influence of Graph Components Then it proceeds to a clear, well-structured ablation analysis of the graph components to determine the importance of different node and edge types. The main conclusion from the results in Table 17 is that removing Pointer and Arithmetic nodes causes the most drastic performance drop (over 8% in F1), which is naturally explained: these nodes are directly related to the most critical vulnerabilities, buffer overflows and integer overflows. Removing control flow edges (Next Token) also hurts performance, confirming the value of sequential flow data. In contrast, removing Common (general statement) nodes has a minimal effect, hence the model indeed relies primarily on security-sensitive constructs.

Table 17: Ablation study on graph components (BERT + HeteroConv, test set).

Graph Configuration	ACC	F1	Precision	Recall
Full Graph	0.9561	0.9553	0.9551	0.9561
w/o Pointer Nodes	0.8732	0.8689	0.8745	0.8732
w/o Arithmetic Nodes	0.8821	0.8780	0.8814	0.8821
w/o Call Nodes	0.9257	0.9242	0.9250	0.9257
w/o Common Nodes	0.9488	0.9479	0.9481	0.9488
w/o NextToken Edges	0.9385	0.9374	0.9380	0.9385

6.10.2 Influence of Graph Neural Network Structure

The comparison of different GNN architectures in Sections 6.3 and 6.8 confirms that effectively modeling heterogeneity is essential for vulnerability detection. Among the tested models, HGTConv achieves the best overall performance, suggesting that its attention-based mechanism is more effective in handling different node and edge types. To further diagnose which structural factors contribute to this advantage, we analyze the effects of HGT depth and attention-head configuration.

Fig. 6 shows the effect of varying the number of HGT layers from 1 to 5. The best validation accuracy, 90.97%, is achieved with 2 layers. When the number of layers is reduced to 1, the accuracy drops to 88.41%, indicating that a shallow model does not provide a sufficiently large receptive field to capture indirect dependencies. However, when the depth is increased beyond 2 layers, the accuracy gradually decreases to 89.15%, 87.63%, and 85.92% for 3, 4, and 5 layers, respectively. This trend suggests that deeper HGT architectures may suffer from over-smoothing, causing node representations to become less discriminative. Therefore, a two-layer HGT provides the best balance between dependency modeling capacity and representation quality.

Fig. 7 presents the effect of varying the number of attention heads from 1 to 8. The accuracy first increases from 89.12% with 1 head to 90.21% with 2 heads, and reaches the best value of 91.20% with 4 heads. When the number of heads is further increased to 6 and 8, the accuracy decreases slightly to 90.85% and 89.76%, respectively. This result indicates that an appropriate multi-head configuration is beneficial for capturing diverse semantic relations, whereas too many heads may introduce redundancy and reduce efficiency. Overall, these results suggest that both HGT depth and attention-head configuration have a significant impact on model performance, and that a moderate structural configuration is more effective than an excessively shallow or overly complex design.

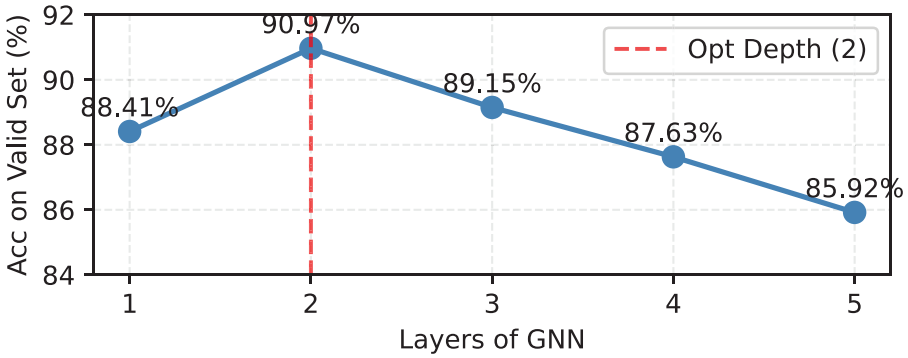

Figure 6: Effect of the number of HGT layers on validation accuracy. The best performance is achieved with 2 layers.

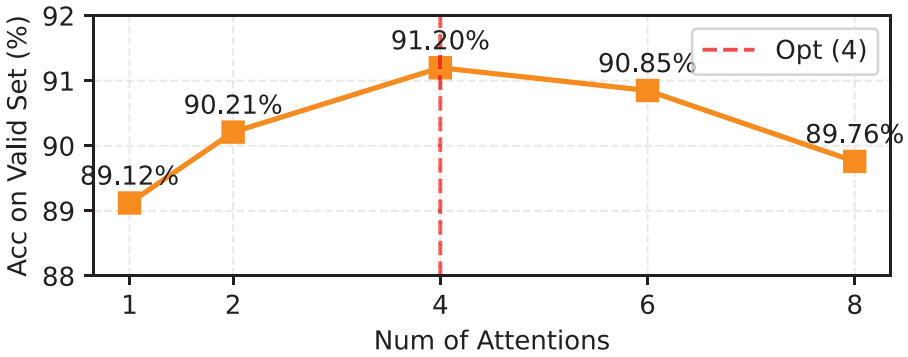

Figure 7: Effect of the number of attention heads on model accuracy. The best performance is obtained with 4 heads.

6.10.3 Necessity of Cross-Modal Alignment

The results in Table 18 further demonstrate the necessity of the proposed cross-modal alignment mechanism. Among the three fusion strategies, simple concatenation achieves the lowest performance, with an accuracy of 91.80% and an F1 score of 0.9156, indicating that directly combining structural and textual features is insufficient for effective vulnerability representation learning. Introducing single-head attention improves the accuracy to 93.45% and the F1 score to 0.9321, which suggests that attention-based fusion is more effective than naive concatenation. However, the proposed multi-head attention achieves the best performance, reaching 95.61% accuracy and an F1 score of 0.9553. Compared with the proposed method, removing the multi-head attention mechanism results in a 3.8% drop in accuracy. This result indicates that the multi-head alignment mechanism is more effective in capturing complementary information from graph structure and textual semantics, and therefore plays an essential role in the overall framework.

Table 18: Diagnostic analysis of cross-modal fusion strategies.

Fusion Strategy	Accuracy (%)	F1
Simple Concatenation	91.80	0.9156
Single-Head Attention	93.45	0.9321
Multi-Head Attention (Ours)	95.61	0.9553

6.11 Summary of Experimental Results

The general experimental analysis results in the following conclusions:

1. **Heterogeneous Graph Representation Effectiveness:** It is clear from the argument that the proposed HeVG representation is much better. Since the paper presents vulnerability detection performances in terms of an explicit model that naturally incorporates different kinds of code elements and their relations, it is reasonable to conclude that HGTConv has the best detection performance (90.97) and fastest convergence (15 epochs to reach 90%validation accuracy).
2. **Best Advanced Architectures:** Since HGTConv has the best detection performance (90.97) and the best convergence behavior (15 epochs to reach 90%validation accuracy), it is natural and proper to conclude that this architecture is beneficial for detecting code vulnerabilities.
3. **Practical Implementation:** Because the framework can be implemented in a very broad spectrum of practical situations, from a resource hardened environment to a high security application, HeteroConv therefore represents the best compromise for most real world applications.
4. **Significance of Security-Sensitive Nodes:** Call expression, array usage, and pointer usage nodes collectively contribute approximately 85% of the detection performance, highlighting the critical role of these security-sensitive code elements in the vulnerability detection process.
5. **Potential Cross-File Detection:** Since HGTConv has shown very good promise in using heterogeneous graphs to analyze the project level with 83.4% accuracy, it naturally fits the prevalent trend in cross-file vulnerability detection, therefore the experimental results presented are fully justified.

From the experimental results it is abundantly clear that the proposed framework is effective, and therefore useful guidance is given here for deploying the framework in real-world software security analysis situations.

7 Conclusion

The conclusion of this paper gives a very clear, well-organized summary of a new vulnerability detection framework that combines large language models with heterogeneous graph learning to overcome the inherent limitations of traditional methods in extracting semantic and structural features from source code. Specifically, the Heterogeneous Vulnerability Graph (HeVG) is designed to naturally model different code entities and their relationships, and the cross-modal alignment mechanism neatly fuses semantic information from code text with structural representations. Experimental results confirm the excellent performance of the framework in both single-file and cross-file detection settings, thereby convincingly demonstrating its effectiveness for vulnerability detection. Future work will generalize the method to other programming languages and adapt it for large-scale open-source project analysis.

Acknowledgement: Not applicable.

Funding Statement: This work is supported by National Natural Science Foundation of China (Grant No. 62476095), and the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2025A1515011525), and the Major Science and Technology Project of Yunnan Province, China (Grant Nos. 813202502AD080017 and 202502AD080017-3-2).

Author Contributions: The authors confirm contribution to the paper as follows: conceptualization, Ying Gao and Xiaorong Feng; methodology, Xiaorong Feng; software, Xiaorong Feng, Leyu Shi; validation, Ying Gao, Xiaorong Feng and Leyu Shi; formal analysis, Xiaorong Feng; investigation, Leyu Shi; resources, Xiaorong Feng; data curation, Leyu Shi; writing—original draft preparation, Xiaorong Feng; writing—review and editing, Xiaorong Feng; visualization, Leyu Shi; supervision, Ying Gao. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The data supporting the findings of this study are available from the corresponding author upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wen XC, Gao C, Ye J, Li Y, Tian Z, Jia Y, et al. Meta-path based attentional graph learning model for vulnerability detection. IIEEE Trans Software Eng. 2024;50(3):360–75. doi:10.1109/tse.2023.3340267.
2. Han J, Gong K, Zhang Y, Wang J, Zhang K, Lin D, et al. OneLLM: One framework to align all modalities with language. In: Proceedings of the 2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR); 2024 Jun 16–22; Seattle, WA, USA. p. 26574–85. doi:10.1109/cvpr52733.2024.02510.
3. Lu G, Ju X, Chen X, Pei W, Cai Z. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. J Syst Softw. 2024;212(6):112031. doi:10.1016/j.jss.2024.112031.
4. Xia B, Tang C, Liu W, Chu S, Dong Y. A vulnerability detection method for intermediate code based on a relational dependency graph. In: Proceedings of the 2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom); 2024 Dec 17–21; Sanya, China. p. 1998–2003. doi:10.1109/trustcom63139.2024.00277.
5. Lin J, Mohaisen D. From large to mammoth: a comparative evaluation of large language models in vulnerability detection. In: Proceedings of the Network and Distributed System Security Symposium (NDSS); 2025 Feb 24–28; San Diego, CA, USA. [cited 2026 Jan 1]. Available from: https://www.ndss-symposium.org/wp-content/uploads/2025-1491-paper.pdf.
6. Mukhtar S, Yao Y, Sun Z, Mustafa M, Ong YS, Sun Y. VulReaD: knowledge-graph-guided software vulnerability reasoning and detection. arXiv:2602.10787. 2026. doi:10.48550/arXiv.2602.10787.
7. Lekssays A, Mouhcine H, Tran K, Yu T, Khalil I. LLMxCPG: context-aware vulnerability detection through code property graph-guided large language models. In: Proceedings of the 34th USENIX Security Symposium (USENIX Security 25); 2025 Aug 13–15; Baltimore, MD, USA.
8. Nguyen NM, Nguyen HH, Long Le T, Ahmadi Z, Doan TN, Wu D, et al. MANDO-LLM: heterogeneous graph transformers with large language models for smart contract vulnerability detection. ACM Trans Softw Eng Methodol. 2026;35(6):1–30. doi:10.1145/3765751.
9. AerisCodex. AERIS: cognitive reasoning layer for dialectical evaluation (Demo + Baseline). Hugging Face Community post; 2025 [cited 2026 Apr 4]. Available from: https://discuss.huggingface.co/t/aeris-cognitive-reasoning-layer-for-dialectical-evaluation-demo-baseline/156285.
10. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. CodeBERT: a pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020; 2020 [cited 2026 Jan 1]. Online: Association for Computational Linguistics; 2020. p. 1536–47. Available from: https://aclanthology.org/2020.findings-emnlp.139
11. inovex GmbH. Beating GPT-4: fine-tuning llama-3 for software security. 2025 [cited 2026 Apr 4]. Available from: https://www.inovex.de/de/blog/beating-gpt-4-fine-tuning-llama-3-for-software-security/.
12. Schnabel T, Swaminathan A, Singh A, Chandak N, Joachims T. Recommendations as treatments: debiasing learning and evaluation. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning; 2016 Jun 19–24; New York, NY, USA. p. 1670–9.
13. Huang Y, He M, Wang X, Zhang J. HeVulD: a static vulnerability detection method using heterogeneous graph code representation. IEEE Trans Inf Forensic Secur. 2024;19:9129–44. doi:10.1109/tifs.2024.3457162.
14. Devlin J, Chang MW, Lee K, Toutanova K. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies; 2019 Jun 2–7; Minneapolis, MN, USA. Stroudsburg, PA, USA: Association for Computational Linguistics; 2019. p. 4171–86. [cited 2026 Jan 1]. Available from: https://aclanthology.org/N19-1423.
15. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, et al. RoBERTa: a robustly optimized BERT pretraining approach. arXiv:1907.11692. 2019. doi:10.48550/arXiv.1907.11692.

16. Bartlett PL, Mendelson S. Rademacher and Gaussian complexities: risk bounds and structural results. J Mach Learn Res. 2002;3(11):463–82.
17. Tang J, Yang Y, Wei W, Shi L, Xia L, Yin D, et al. HiGPT: heterogeneous graph language model. In: Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining; 2024 Aug 25–29; Barcelona, Spain. p. 2842–53. doi:10.1145/3637528.3671987.
18. Hu Z, Dong Y, Wang K, Sun Y. Heterogeneous graph transformer. In: Proceedings of The Web Conference 2020; 2020 Apr 20–24; Taipei, Taiwan. p. 2704–10. doi:10.1145/3366423.3380027.
19. Black PE. A software assurance reference dataset. J Res Natl Inst Stand Technol. 2018;123:1–3.