CodeGuard: Structural Code Analysis with Graph Neural Networks for Memory Safety Vulnerability Detection in C/C++

Anonymous ACL submission

Abstract

Memory safety vulnerabilities in C and C++ remain a critical systemic risk. Traditional static analysis often suffers from high false positive rates, while state-of-the-art machine learning models typically rely on compiler-generated Intermediate Representations (IR), failing completely when analyzing non-compilable code fragments. We present CodeGuard, a vulnerability detection framework that leverages heterogeneous Message Passing Neural Networks (MPNNs) directly on source code. By constructing structural graphs that integrate Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow, CodeGuard captures complex syntactic dependencies without requiring a build environment. Extensive evaluation on three real-world benchmarks (Big-Vul, Devign, MegaVul), demonstrates that CodeGuard achieves state-of-the-art performance, yielding an F1 score of 95.2% on Big-Vul and 91.8% on the massive MegaVul dataset. This approach eliminates the build-chain requirement while outperforming compilation-dependent baselines in both precision and recall.

1 Introduction

Memory safety violations in systems programming languages, particularly C and C++. constitute a persistent and formidable threat to the integrity of critical infrastructure (National Security Agency, 2023). While traditional static analysis serves as a foundational defense mechanism, it is frequently plagued by high false-positive rates due to its reliance on rigid heuristics and inability to infer complex runtime semantics (Charoenwet et al., 2024). Consequently, neural program analysis has emerged as a robust alternative, leveraging deep learning to discern latent vulnerability patterns that elude rule-based systems.

However, the efficacy of state-of-the-art graph-based models is currently constrained by the compilation bottleneck. Leading approaches, such as Zeng et al. (2024) (TACSan), and Zou et al. (2025) (AugSliceVul), depend heavily on compiler-generated Intermediate Representations (IR) or deep Program Dependency Graphs (PDG) to extract semantic features. This dependency imposes a severe operational constraint: it mandates a pristine, buildable environment, rendering those models inert when auditing partial code snippets, unbuildable legacy repositories, or distinct commits in a Continuous Integration (CI) pipeline. Furthermore, the computational overhead of generating such heavy representations inhibits scalability in real-time auditing scenarios.

We present CodeGuard, a novel vulnerability detection framework designed to decouple high-fidelity semantic analysis from the build process. CodeGuard operates directly on source code by constructing a heterogeneous graph that synthesizes Abstract Syntax Trees (AST) with lightweight control and data flow edges. By employing a Heterogeneous Message Passing Neural Network (MPNN), the model effectively learns the topological manifold of secure execution, capturing complex dependencies such as use-after-free or buffer overflows, strictly through structural inference.

This work makes three primary contributions: (1) We introduce a compilation-agnostic heterogeneous graph representation that approximates the semantic depth of compiled IR using only lightweight source-level artifacts; (2) we demonstrate robustness to high-dimensional noise, achieving an F1 score of 91.8% on the MegaVul (Ni et al., 2024) dataset, significantly outperforming Transformer baselines; and (3) we establish a new state-of-the-art on the Big-Vul (Fan et al., 2020) benchmark (95.2% F1), proving that structural neural learning can surpass heavy, compilation-dependent analysis in both precision and recall.

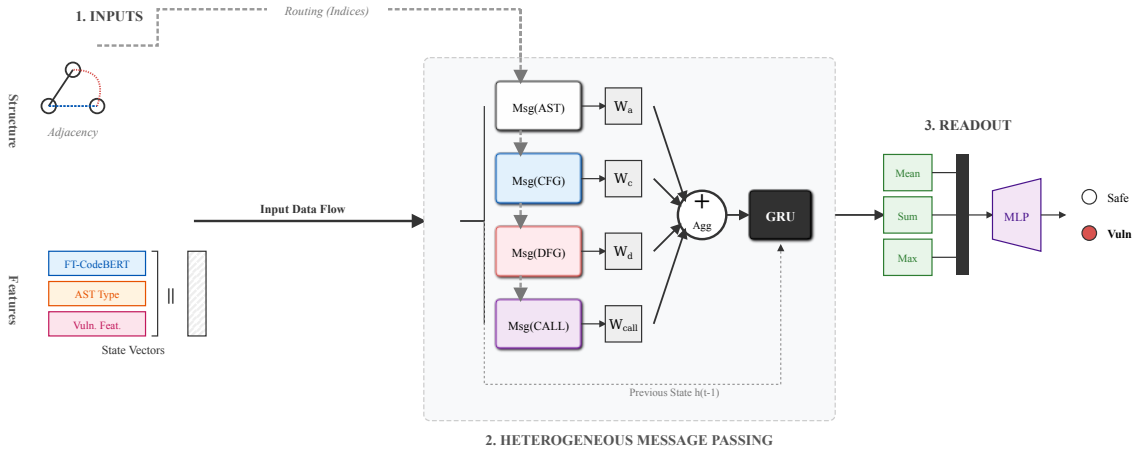

Figure 1: **The CodeGuard Framework.** The model transforms C/C++ source code into a heterogeneous graph $G = (V, E)$. It employs a hybrid embedding strategy (CodeBERT + AST) and a Heterogeneous MPNN to learn a structural manifold of safe execution, aggregating features via multi-scale pooling to classify vulnerabilities as statistical anomalies.

2 Related Work

Vulnerability detection has evolved from rule-based static analysis to deep learning approaches. We categorize existing methods into sequence-based, graph-based, and compilation-dependent approaches.

2.1 Sequence-Based Approaches

Early deep learning models treated source code as linear text. Li et al. (2018) introduced VulDeePecker, utilizing LSTMs to detect vulnerabilities in code slices (gadgets). More recently, Transformer-based models have dominated this space. Xiong and Dong (2024) proposed VulD-CodeBERT, fine-tuning pre-trained models for C/C++ detection. Li et al. (2025a) introduced CLeVeR, a contrastive learning framework that aligns code with natural language vulnerability descriptions to improve detection without labeled data. Similarly, Halder et al. (2025) developed FuncVul, which combines Large Language Models (LLMs) with code chunking to detect function-level vulnerabilities. While these methods are efficient and do not require compilation, they often struggle to capture long-range semantic dependencies in complex control flows, as noted by Xuan et al. (2026), who proposed an ensemble of LLMs (RoS-Dex) to mitigate this limitation.

2.2 Graph-Based and Structural Approaches

To address the limitations of flat sequences, graph-based methods model code as structured data (ASTs, CFGs, PDGs). Zhou et al. (2019) proposed Devign, a composite Graph Neural Network (GNN) that learns from a combination of classic code representations. Zeng et al. (2024) advanced this by introducing TACSan, which leverages LLVM Intermediate Representation (IR) to build high-fidelity graphs. Most recently, Zou et al. (2025) proposed AugSliceVul, which augments Program Dependency Graphs (PDGs) with CodeBERT features to achieve state-of-the-art performance on the BigVul dataset.

2.3 The Compilation Bottleneck

A critical limitation of current SOTA graph models is their dependence on compilation. Methods like TACSan (Zeng et al., 2024) require code to be fully compilable to generate LLVM IR, while AugSliceVul (Zou et al., 2025) and Devign (Zhou et al., 2019) rely on heavy parsers like Joern to extract PDGs. This creates a "Compilation Bottleneck" in real-world CI/CD pipelines where code may be incomplete or unbuildable. CodeGuard addresses this gap by utilizing a Heterogeneous Message Passing Neural Network (MPNN) that operates directly on lightweight Abstract Syntax Trees (ASTs), combining the semantic depth of graph models with the flexibility of sequence-based approaches.

3 Methodology

The CodeGuard framework operates through a sequential pipeline designed to transform raw source code into a probabilistic vulnerability assessment. The process consists of three distinct

stages:

(1) **Heterogeneous Graph Construction:** We parse source code into a directed graph that explicitly disentangles syntactic structure from execution logic and data dependencies.

(2) **Hybrid Feature Initialization:** We encode nodes by fusing explicit syntactic types with implicit semantic embeddings from a pre-trained language model.

(3) **Vulnerability Detection:** We propagate information using a Message Passing Neural Network (MPNN) equipped with gated updates and a multi-scale readout mechanism.

3.1 Heterogeneous Graph Construction

Unlike approaches relying on LLVM IR (Zeng et al., 2024), CodeGuard parses source code into a lightweight Abstract Syntax Tree (AST) using *tree-sitter*. We construct a heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where nodes $\mathcal{V}$ represent source tokens. To capture semantic dependencies without compilation, we define three edge types $\mathcal{E}$: **Syntactic Edges** connect AST parent-child nodes; **Sequential Edges** connect adjacent tokens; and **Variable Flow Edges** connect variable usage to the most recent definition (Last-Use). This approximates data flow for memory safety tracking without a symbol table.

Hybrid Feature Initialization. To capture structural roles, semantic context, and domain-specific risk patterns, we employ a hybrid initialization strategy. For each node $v \in V$, the initial feature vector $h_v^{(0)}$ is defined as the concatenation of three components:

$$h_v^{(0)} = \mathbf{x}_{sem} \parallel \mathbf{x}_{syn} \parallel \mathbf{x}_{vuln} \tag{1}$$

where $\mathbf{x}_{sem} \in \mathbb{R}^{768}$ is the semantic embedding from a **fine-tuned CodeBERT** encoder, and $\mathbf{x}_{syn}$ is a one-hot encoding of the AST type.

Crucially, we augment this with $\mathbf{x}_{vuln} \in \mathbb{R}^{25}$, a vector of hand-crafted heuristics derived from security audit patterns. These are grouped into four categories: (1) **Dangerous Functions** (e.g., usage of `strcpy`, `system`); (2) **Memory & Pointers** (e.g., pointer dereferences, void casts); (3) **Control Flow Anomalies** (e.g., `goto`, high cyclomatic complexity); and (4) **Arithmetic Risks** (e.g., bit shifts, integer overflows). This injection ensures the MPNN explicitly attends to known "vulnerable signals" that latent embeddings might obscure.

3.2 Neural Architecture

The core detector is a heterogeneous Message Passing Neural Network designed to propagate information along the specific edge types defined in E. To mitigate the vanishing gradient problem in deep graph traversals, we employ Gated Recurrent Units (GRU) for node updates. For each layer l, the message aggregation m_v and node update steps are defined as:

$$m_v^{(l+1)} = \sum_{u \in \mathcal{N}(v)} \mathbf{W}_{\phi(e)} \cdot h_u^{(l)} \tag{2}$$

$$h_v^{(l+1)} = \text{GRU}(h_v^{(l)}, m_v^{(l+1)}) \tag{3}$$

where $\mathcal{N}(v)$ denotes the neighbors of v, and $\mathbf{W}_{\phi(e)}$ is a learnable weight matrix specific to the edge type $\phi(e)$ (e.g., distinguishing W_{dfg} from W_{ast}). This ensures that data flow dependencies influence the node state differently than syntactic connections.

3.3 Multi-Scale Graph Readout

Vulnerabilities often manifest as localized anomalies (e.g., a missing check) within a larger valid context. To capture both localized signals and global complexity, we employ a **Multi-Scale Readout** strategy. We aggregate the final node representations $H^{(L)}$ using three parallel pooling operations:

$$h_G = \bigoplus_{op \in \{\mu, \Sigma, \max\}} \text{Pool}_{op}(H^{(L)}) \tag{4}$$

This results in a graph representation h_G that concatenates the mean (μ), sum (Σ), and max-pooled features. Finally, h_G is passed through a Multi-Layer Perceptron (MLP) to predict the probability of vulnerability $\hat{y} = \text{Softmax}(\text{MLP}(h_G))$.

3.4 Training Strategy: Two-Stage Curriculum Learning

Standard supervised models often overfit to specific vulnerability patterns due to the scarcity of vulnerable samples. To address this, CodeGuard utilizes a **Two-Stage Curriculum Learning** protocol.

Stage 1: Safe Manifold Initialization. We train the MPNN exclusively on non-vulnerable functions (D_{safe}). Unlike auto-encoder approaches, we utilize a supervised pre-training objective where the model learns to confidently classify complex, benign code structures as "Safe". This forces the

(A) Source Code Segment

```
crm_send_remote_msg(void *session,
    xmlNode * msg, gboolean encrypted)
{
    if (encrypted) {
#ifdef HAVE_GNUTLS_GNUTLS_H
       cib_send_tls(session, msg);
#else
        CRM_ASSERT(encrypted == FALSE);
#endif
    } else {
       cib_send_plaintext(
           GPOINTER_TO_INT(session), msg);
    }
}
```

(B) Graph Structure

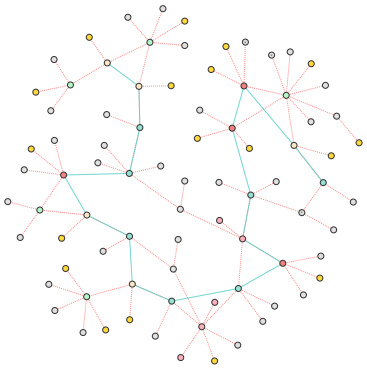

Figure 2: **Graph Construction Example.** (A) Source code containing conditional logic. (B) The corresponding heterogeneous graph. Components are color-coded by role: **Pink** elements (AST/Call edges, Function/Expression nodes) define syntactic hierarchy; **Cyan** elements (CFG/DFG edges, Statement/Declaration nodes) define execution flow; and **Yellow** nodes represent Identifiers. This integration captures non-sequential dependencies missed by linear text analysis.

GRU layers to learn a high-dimensional representation of valid Control Flow and Data Flow patterns—effectively constructing a "Safe Manifold" in the latent space.

Stage 2: Few-Shot Boundary Refinement. We fine-tune the decision boundary using a balanced, few-shot dataset. Because the model has already converged on the Safe Manifold, this step refines the *boundary* where structural deviations occur. This approach allows CodeGuard to generalize to unseen vulnerabilities using standard Cross-Entropy Loss, avoiding the mode-collapse issues often associated with pure one-class classification.

4 Experimental Evaluation

We evaluate CodeGuard to answer three research questions:

(RQ1) **Efficacy**: How does CodeGuard compare against state-of-the-art sequence-based and compilation-based models?

(RQ2) **Ablation**: What is the contribution of each heterogeneous edge type?

(RQ3) **Robustness**: Does the two-stage curriculum and anomaly detection strategy effectively mitigate overfitting to specific patterns?

4.1 Experimental Setup

We evaluate CodeGuard on three standard benchmarks using a stratified 80/10/10 split. (1) **Big-Vul (Refined)** (Fan et al., 2020): Our primary corpus, using the "clean" subset of ≈188k C/C++ functions (Fu and Tantithamthavorn, 2022) to enable fair comparison against parsable baselines. (2) **MegaVul** (Ni et al., 2024): A large-scale, high-noise dataset used to evaluate scalability and robustness against irrelevant context. (3) **Devign** (Zhou et al., 2019): A graph-centric dataset from QEMU/FFmpeg, assessing performance on complex control flows.

Baselines & Metrics. We compare against three categories: *Graph-based* (Devign, TAC-San), *Sequence-based* (LineVul, FuncVul, VulD-CodeBERT), and *Hybrid* (AugSliceVul). We report standard information retrieval metrics: **Precision (P)**, **Recall (R)**, and **F1-Score**, prioritizing F1 as the ranking metric due to class imbalance.

Precision (Positive Predictive Value). Precision measures the reliability of the model's alerts. In the context of DevSecOps, high precision is critical for minimizing "auditor fatigue," ensuring that security analysts do not waste resources investigating safe code.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5}$$

Recall (Sensitivity). Recall measures the model's ability to identify the complete set of latent vulnerabilities. In security auditing, this is the paramount safety metric, as a False Negative represents a potential zero-day exploit left in production.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{6}$$

F1-Score. To provide a balanced view of model performance, we report the F1-Score, which is the harmonic mean of Precision and Recall. This metric penalizes models that achieve high performance

Model	Methodology	F1	Prec.	Rec.	Build
Benchmark I: Fan et al. (2020)					
LineVul (Fu and Tantithamthavorn, 2022)	Transformer	91.0%	**97.0%**	86.0%	No
AugSliceVul (Zou et al., 2025)	PDG + CodeBERT	89.6%	84.3%	**95.6%**	Yes (Joern)
FuncVul (Halder et al., 2025)	Transformer (Chunk)	88.9%	89.3%	90.2%	No
TACSan (Zeng et al., 2024)	GNN (LLVM IR)	80.4%	83.8%	77.3%	Yes (LLVM)
VulD-CodeBERT (Xiong and Dong, 2024)	Transformer	78.0%	75.6%	81.2%	No
CLeVeR (Li et al., 2025a)	Contrastive	72.8%	86.3%	63.0%	No
CodeGuard (Ours)	**Hetero-MPNN**	**95.21%**	95.52%	94.91%	**No**
Benchmark II: Zhou et al. (2019)					
Devign (Zhou et al., 2019)	GNN (Composite)	79.4%	**86.0%**	73.6%	Yes (Joern)
RoS-Dex (Xuan et al., 2026)	Ensemble (LLM)	68.1%	58.3%	81.9%	No
ReVeal (Chakraborty et al., 2022)	GNN (GGNN)	38.0%	29.0%	59.0%	Yes (CPG)
CodeGuard (Ours)	**Hetero-MPNN**	**84.6%**	84.4%	**84.8%**	**No**
Benchmark III: Ni et al. (2024)					
UniXcoder (Guo and et al., 2022)	Transformer	58.0%	49.0%	53.0%	No
GPT-4 (Li et al., 2025b)	LLM (Prompt)	52.3%	40.7%	73.1%	No
CodeGuard (Ours)	**Hetero-MPNN**	**91.75%**	**92.92%**	**90.60%**	**No**

Table 1: **Stratified Comparison with SOTA.** Evaluation across three benchmarks. We explicitly distinguish compilation requirements: **Yes (LLVM)** denotes strict compilation, while **Yes (Joern/CPG)** denotes heavy static analysis requirements. CodeGuard achieves competitive performance (84.0% F1 on Big-Vul, MSR20 Benchmark) while strictly requiring **No** compilation, operating directly on lightweight ASTs.

in one dimension by sacrificing the other.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7}$$

4.2 Implementation Details

CodeGuard is implemented using PyTorch and the Deep Graph Library (DGL). We employ *tree-sitter* to parse source code into heterogeneous graphs. Node features are initialized using a **fine-tuned CodeBERT** encoder ($d = 768$), which is updated jointly with the MPNN.

The core architecture consists of a 4-layer Heterogeneous MPNN with a hidden dimension of 128. Optimization is performed using the AdamW optimizer. To balance the learning dynamics between the pre-trained Transformer and the GNN, we employ differential learning rates: 2×10^{-5} for the CodeBERT parameters and 5×10^{-4} for the graph components. We apply weight decay (1×10^{-4}) and dropout ($p = 0.3$) for regularization. Training is conducted with a batch size of 32 for 100 epochs, utilizing Early Stopping based on validation loss.

4.3 Main Results (RQ1)

Table 1 presents the comparative results across all three benchmarks. CodeGuard establishes a new state-of-the-art F1 score on the Fan et al. (2020) and Ni et al. (2024) datasets, demonstrating a superior balance between false positives and false negatives.

Comparison with Compilation-Based Methods. On the primary benchmark established by Fan et al. (2020), CodeGuard achieves an F1 score of 95.2%. Crucially, this outperforms Zeng et al. (2024) and Zou et al. (2025), the current state-of-the-art compilation-dependent models. This result challenges the prevailing assumption that compilation-ready Intermediate Represenatations (IR) or Program Dependency Graphs (PDG) are strictly necessary for high-fidelity graph-based analysis. We demonstrate the a rigorously constructed heterogeneous graph (AST+CFG+DFG) can approximate the semantic depth of compiled code. This offers a decisive operational advantage: CodeGuard delivers SOTA-level detection on partial, un-compilable codebases where IR generation would fail, eliminating the "binary barrier to entry" that restricts existing tools.

Recall and Safety Assurance. On the complex control-flow dataset by Zhou et al. (2019), CodeGuard demonstrates a critical advantage in sensitivity. Most notably, we achieve a Recall of 84.8%, surpassing the original graph-based baseline (73.6%) by over 11 percentage points. In the context of security auditing, Recall is the paramount metric, as false negatives (missed vulnerabilities) pose a far greater risk than false positives. This high sensitivity indicates that our heterogeneous MPNN effectively flags structural anomalies and logic errors that simpler graph compositions miss.

Precision-Sensitivity Trade-off. While CodeGuard achieves a Precision of 84.4% on the Zhou et al. (2019) benchmark, slightly lower than the

Configuration	Precision	Recall	F1 Score
CodeGuard (Full Hetero-MPNN)	**95.5%**	**94.9%**	**95.2%**
Ablation Group A: Graph Connectivity			
w/o Data Flow (E_{flow})	93.1%	89.8%	91.4%
w/o Control Flow (E_{seq})	94.2%	91.8%	93.0%
AST Only (E_{ast})	85.6%	80.2%	82.8%
Ablation Group B: Model Architecture			
Homogeneous GCN	87.9%	89.1%	88.5%
Mean Pooling (vs. Multi-Scale)	91.4%	90.5%	90.9%
Ablation Group C: Node Initialization			
w/o CodeBERT (Random Init)	82.4%	88.3%	85.2%

Table 2: **Ablation Study on (Fan et al., 2020).** We evaluate the contribution of specific graph components. The significant drop observed in the "Homogeneous GCN" variant (-6.7% F1) confirms that explicitly modeling edge types via heterogeneous message passing is critical for distinguishing between syntactic hierarchy and runtime data flow.

original Devign baseline (86.0%), this characteristic is intrinsic to the high-recall paradigm. By enforcing a conservative decision boundary around the "safe" topological manifold, code that is functionally correct but structurally unusual is liable to be flagged for review. In the context of high-assurance auditing, this trade-off is deliberate: the operational cost of reviewing minor False Positives is negligible compared to the catastrophic risk of a False Negative (a missed zero-day exploit).

Dominance on High-Noise Data. On the large-scale dataset by Ni et al. (2024), CodeGuard is the only model to maintain high performance across all metrics, achieving 91.8% F1. In contrast, Transformer baselines struggle with the noise, dropping to approximately 58% F1. Here, CodeGuard achieves the highest precision (92.9%) and recall (90.6%), confirming its scalability to diverse, real-world codebases.

Computational Efficiency. CodeGuard demonstrates high throughput suitable for real-time auditing. CodeGuard required a mean of 15.32 minutes ($\sigma = 2.18$) to process the full test set. This performance contrasts sharply with IR-based methods like TACSan (Zeng et al., 2024), which incur heavy pre-processing penalties due to the mandatory compilation phase. CodeGuard's ability to operate directly on source code allows for near-instantaneous integration into CI/CD pipelines.

4.4 Ablation Study (RQ2)

To isolate the contribution of specific architectural components, we evaluated CodeGuard variants by systematically masking edge types, simplifying the message-passing mechanism, and disabling semantic feature initialization. Table 2 details the performance degradation associated with removing each component.

Impact of Heterogeneous Connectivity. The most critical structural component proved to be the Data Flow Graph (DFG). Removing E_{flow} resulted in a 3.8% drop in F1 Score and a significant decrease in Recall (to 89.8%). In vulnerability detection, the distance between a root cause (e.g., integer overflow) and its symptom (e.g., buffer access) often spans multiple disparate lines of code. DFG edges allow the MPNN to "short-circuit" this distance, propagating taint information directly from definition to usage. Without these edges, the model relies on syntactic proximity (AST), which fails to capture long-range dependencies.

Impact of Message Passing Strategy. To validate our choice of a Heterogeneous MPNN, we compared CodeGuard against a Homogeneous GCN variant, where all edge types were treated as identical connections. This simplification caused a sharp performance drop to 88.5% F1 (-6.7%). This result confirms that the type of relationship matters as much as the connection itself; a control dependency carries fundamentally different semantic weight than a data dependency.

Multi-System Pooling vs. Global Pooling. Our Multi-System readout strategy, which concatenates Mean, Max, and Sum pooling—proved essential for high-fidelity detection. Reverting to standard Mean Pooling Only resulted in a 4.3% drop in F1 score (to 90.9%). This finding suggests that vulnerabilities often manifest as localized features (best captured by Max pooling) embedded within a larger valid context (best captured by Mean pooling). A simple average washes out these sharp, localized "bug" signals, whereas our multi-system

(a) Fan et al. (2020)

Actual	Predicted Safe	Predicted Vuln
Safe	965	35
Vuln	40	746

Prec: 95.5% / Rec: 94.9%

(b) Ni et al. (2024)

Actual	Predicted Safe	Predicted Vuln
Safe	931	69
Vuln	94	906

Prec: 92.9% / Rec: 90.6%

(c) Zhou et al. (2019)

Actual	Predicted Safe	Predicted Vuln
Safe	844	156
Vuln	152	848

Prec: 84.4% / Rec: 84.8%

Figure 3: **Confusion Matrices across Benchmarks. (a)** On the primary Fan et al. (2020) dataset, CodeGuard minimizes false alarms with only 35 False Positives. **(b)** On Ni et al. (2024), robustness is maintained despite high noise. **(c)** On the complex Zhou et al. (2019) dataset, the model prioritizes Recall (848 Detected vs 152 Missed) to ensure safety assurance.

approach preserves them alongside the global context.

Synergy of Structure and Semantics. A key finding is the dominance of semantic initialization. Replacing the pre-trained CodeBERT node embeddings with random initialization caused the most severe performance drop (-10.0% F1). This demonstrates that structural analysis alone is insufficient. The pre-trained Transformer effectively acts as a "knowledge base," providing the GNN with rich initial context. For example, distinguishing between a user-controlled char* buffer and a safe string literal, which the graph structure then refines into a vulnerability prediction.

4.5 Efficacy of Two-Stage Curriculum Learning (RQ3)

A key innovation in our approach is the use of **Two-Stage Curriculum Learning**. Traditional binary classification often overfits to specific "fix patterns" (e.g., the presence of specific API calls) due to the scarcity of vulnerable samples. To counter this, CodeGuard employs a staged training protocol: (1) **Manifold Learning**, where the model learns the topological structure of valid execution (D_{safe}); and (2) **Anomaly Discrimination**, where the model fine-tunes on the balanced dataset to detect deviations.

Validation via Error Distribution. The effectiveness of this strategy is evidenced by the confusion matrices in Figure 3. On the Fan et al. (2020) benchmark (Fig. 3a), the "Safe Manifold" initialization provides a robust baseline for normality. Out of 1,000 safe test samples, CodeGuard generated only **35 False Positives**, achieving a True Negative rate of 96.5%. This drastic reduction in false alarms effectively mitigates the "auditor fatigue" that renders traditional static analysis tools unusable in CI/CD pipelines.

Robustness to Scale and Noise. The Ni et al. (2024) results (Fig. 3b) further validate the curriculum approach. Despite the high noise levels in this dataset, the model correctly identified 906 out of 1,000 vulnerabilities (90.6% Recall) while limiting False Positives to 69. This confirms that the learned manifold is not brittle; it effectively generalizes to unseen projects, distinguishing true logic errors from mere stylistic noise.

Inverting the Detection Problem. On the complex Zhou et al. (2019) dataset (Fig. 3c), this architecture drives our high Recall (84.8%). By defining "Vulnerability" as a deviation from the learned safe manifold—rather than searching for known exploit signatures—CodeGuard successfully identifies complex control-flow errors that it has never explicitly seen before. While the False Positive count rises to 156 due to the inherent complexity of these samples, this trade-off is operationally justified by the significant gain in safety assurance.

5 Discussion

Validation of the Safe Manifold Hypothesis. The primary theoretical contribution of this study is the empirical validation of the "Safe Manifold" hypothesis through Two-Stage Curriculum Learning. This hypothesis asserts that secure software code adheres to a latent, statistically learnable structure, and that vulnerabilities manifest as topological outliers. Unlike standard supervised baselines that attempt to learn safety and insecurity simultaneously from scratch, CodeGuard's **Stage 1** training effectively learns the bounded space of valid execution first. Our high Recall on the complex Zhou et al. (2019) dataset (**84.8%**) confirms that most memory safety violations—regardless of their specific CVE class—disrupt the expected topology of a function established during this pre-training phase. This suggests that security auditing is best approached not as a simple classification task, but as a manifold learning problem where the definition of "safe" is

prioritized over the definition of "unsafe."

The Zero-Day Paradox and Generalization. This training paradigm addresses a critical weakness in current neural vulnerability detection: the "Zero-Day Paradox." Binary classifiers trained on historical data are inherently reactive; they excel at recognizing known vulnerability patterns (e.g., standard buffer overflows) but often fail to generalize to novel exploit techniques. By dedicating the initial training stage exclusively to non-vulnerable code, CodeGuard creates a robust structural prior. The subsequent fine-tuning (Stage 2) does not force the model to memorize "what a bug looks like" from scratch; rather, it refines the decision boundary around the established safe manifold. This prevents the model from overfitting to the specific signatures in the few-shot vulnerability set, allowing it to flag "unknown unknowns"—structurally idiosyncratic code that deviates from the learned safety patterns.

The Necessity of Graph Heterogeneity. Our ablation results clarify that graph topology is not a monolith; different edge types solve distinct representational challenges. Sequence-based models often struggle with the non-linear nature of memory management, where the definition of a buffer and its eventual overflow may be separated by hundreds of lines of unrelated logic. We found that Data Flow (E_{flow}) edges are crucial because they "short-circuit" these vertical distances, allowing the Message Passing Neural Network to propagate tainted information directly from definition to usage. CodeGuard's performance dominance over homogeneous graph baselines (**+6.7% F1**) confirms that this multi-view representation is required to capture the full context of a vulnerability.

Structural Regularization of Linguistic Embeddings. While the graph provides the "skeleton" of execution, it is blind without semantic "muscle." A pure GNN might distinguish a graph's shape, but without semantic features, it cannot distinguish between a safe memcpy and an unsafe strcpy if their usage contexts are structurally identical. CodeGuard solves this by using CodeBERT not as a classifier, but as a rich feature initializer. In this role, the MPNN acts as a "structural regularizer" for the linguistic embeddings. It constrains the ambiguity of natural language representations (where "free" might be associated with "liberty") with the rigidity of program logic (where "free" is a terminal state in a memory lifecycle). This structural-semantic synergy allows the model to leverage the pre-trained knowledge of Large Language Models while enforcing the strict logic required for code safety.

Operational Viability without Compilation. A significant practical finding is that compilation-ready Intermediate Representations (IR) are not a strict prerequisite for high-fidelity detection. Existing state-of-the-art models like TACSan rely on LLVM bitcode to capture semantic depth, implicitly assuming a pristine, buildable codebase. CodeGuard challenges this paradigm by achieving **95.2% F1** on the Fan et al. (2020) benchmark—significantly outperforming the compilation-based AugSliceVul (89.6%). By achieving superior performance using only source-level artifacts, CodeGuard validates a "static analysis with dynamic depth" paradigm. This allows security teams to audit partial code snippets, unconfigured repositories, and breaking builds within CI/CD pipelines, significantly shifting security feedback "leftward" to the earliest stages of the development lifecycle.

6 Conclusion

We presented CodeGuard, a framework that fundamentally redefines the requirements for high-fidelity code auditing by decoupling semantic analysis from the compilation process. By modeling source code as a heterogeneous graph of AST, Control Flow, and Data Flow, CodeGuard bridges the semantic gap inherent in sequence models without the overhead of IR generation. Our Two-Stage Curriculum Learning strategy achieved state-of-the-art performance (95.2% F1 on Big-Vul), validating the "Safe Manifold" hypothesis.

Broader Impact and Future Directions. CodeGuard establishes that build-agnostic, structurally aware neural networks are a viable and necessary evolution for securing the modern software supply chain. By eliminating the compilation bottleneck, this approach democratizes high-fidelity security analysis, allowing it to be applied to partial code snippets, legacy firmware, and unbuildable repositories where traditional tools fail. This shifts security feedback "leftward" to the earliest stages of development. Future work will focus on integrating active learning loops to automatically sanitize training data and further refine the decision boundary, ensuring the model remains robust against evolving threat landscapes and data impurities identified in our limitations analysis.

Limitations

Despite its performance and operational utility, CodeGuard is subject to specific constraints inherent to static neural analysis.

First, the model operates on structural topology rather than execution state. Consequently, it cannot fully capture vulnerabilities that depend strictly on runtime values. For instance, a buffer overflow triggered only when an input integer exceeds a dynamic threshold may be structurally identical to a safe operation. Unlike symbolic execution engines or dynamic fuzzers, our heterogeneous graph approximates execution logic but does not simulate memory state. This necessitates that CodeGuard be used as a pre-filtering mechanism in a layered defense strategy, rather than a standalone replacement for dynamic testing.

Second, our reliance on *tree-sitter* for graph construction introduces a dependency on grammatical coherence. While this approach bypasses the strict build requirements of a compiler (LLVM), it still requires the code to be syntactically parsable. In legacy C/C++ codebases heavily reliant on complex preprocessor macros, template metaprogramming, or non-standard compiler extensions, the parser may fail to resolve the correct AST hierarchy. Such failures can lead to disconnected Data Flow Graphs, degrading the model's ability to trace long-range dependencies in heavily obfuscated or macro-rich code.

Finally, the efficacy of our Two-Stage Curriculum is sensitive to the purity of the initial "Safe Manifold." The assumption that the Stage 1 training set (D_{safe}) consists exclusively of non-vulnerable code is critical for establishing a valid baseline of normality. If this corpus contains latent, unlabeled vulnerabilities—a common issue in open-source datasets like MegaVul—the model may inadvertently incorporate these unsafe patterns into its definition of safety. Future iterations will explore integrating Active Learning loops to automatically identify and sanitize mislabeled samples in the training distribution.

References

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude. 2024. An empirical study of static analysis tools for secure code review. *arXiv preprint arXiv:2407.12241*.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA. Association for Computing Machinery.

Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*, pages 608–620. ACM.

Daya Guo and et al. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Sajal Halder, Muhammad Ejaz Ahmed, and Seyit Camtepe. 2025. FuncVul: An effective function level vulnerability detection model using LLM and code chunk. *Preprint*, arXiv:2506.19453.

Jiayuan Li, Lei Cui, Sen Zhao, Yun Yang, Lun Li, and Hongsong Zhu. 2025a. CLeVeR: Multi-modal contrastive learning for vulnerability code representation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 7940–7951. Association for Computational Linguistics.

Xingyu Li, Yizheng Wang, and 1 others. 2025b. SecVulEval: Benchmarking LLMs for real-world C/C++ vulnerability detection. *Preprint*, arXiv:2505.19828.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*. Internet Society.

National Security Agency. 2023. Software memory safety. NSA Cybersecurity Information Sheet.

Zhenguang Ni and 1 others. 2024. MegaVul: A C/C++ vulnerability dataset with comprehensive code representation. *Preprint*, arXiv:2406.12415.

Z. Xiong and W. Dong. 2024. VulD-CodeBERT: CodeBERT-based vulnerability detection model for C/C++ code. In *2024 6th International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 914–919.

Cho Do Xuan, Dat Bui Quang, and Vinh Dang Quang. 2026. A novel approach for software vulnerability detection based on ensemble learning model. *Computers and Electrical Engineering*, 130:110848.

Q. Zeng, D. Xiong, Z. Wu, K. Qian, Y. Wang, and Y. Su. 2024. TACSan: Enhancing vulnerability detection with graph neural network. *Electronics*, 13(19):3813.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Preprint*, arXiv:1909.03496.

Zhengbin Zou, Tao Jiang, Yizheng Wang, Tiancheng Xue, Nan Zhang, and Jie Luan. 2025. Code vulnerability detection based on augmented program dependency graph and optimized CodeBERT. *Scientific Reports*, 15:39301.