

Rust Vulnerability Classification with WL-Enhanced LLVM-IR-Derived Code Property Graphs

Syeda Jannatul Boshra

Texas A&M University – San Antonio

Young Lee

y1ee@tamusa.edu

Texas A&M University – San Antonio

Research Article

Keywords: Rust, software vulnerability detection, Code Property Graph, LLVM intermediate representation, Weisfeiler–Lehman, XGBoost

Posted Date: October 23rd, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11018699/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.
[Read Full License](#)

Additional Declarations: No competing interests reported.

Rust Vulnerability Classification with WL-Enhanced LLVM-IR-Derived Code Property Graphs

Syeda Jannatul Boshra¹ and Young Lee^{1*}

¹Department of Computational, Engineering and Mathematical Sciences, Texas A&M University—San Antonio, San Antonio, Texas, USA.

*Corresponding author(s). E-mail(s): ylee@tamusa.edu;

Abstract

Rust’s ownership paradigm provides substantial but not absolute safety; vulnerabilities may still arise from unsafe code, flawed logic, concurrency behavior, or insecure dependencies. This paper investigates whether security patterns can be distinguished using Code Property Graphs (CPGs) generated from Rust-produced LLVM Intermediate Representation (LLVM IR). The dataset was produced through a repeatable workflow spanning RustSec-guided source collection, LLVM IR generation, CPG conversion, and JSON validation. After excluding malformed graphs and graphs exceeding the processing limit, the dataset comprised 235 CPGs—145 safe-labeled and 90 vulnerable-labeled. Each CPG was represented as a 182-dimensional hybrid vector combining pooled node attributes, a two-iteration Weisfeiler–Lehman (WL) neighborhood signature, and twenty graph-level descriptors. Three classifiers—XGBoost, Random Forest, and a radial basis function support vector machine—were evaluated on this representation, while a graph convolutional network (GCN) learned directly from CPG structure under five-fold stratified cross-validation partitions. XGBoost achieved the strongest observed performance, with 78.81% balanced accuracy, a receiver operating characteristic area under the curve (ROC–AUC) of 0.9033, and a precision–recall area under the curve (PR–AUC) of 0.8603. Feature-component analysis shows an observed improvement when WL features are added to pooled node attributes for XGBoost, while pooled node attributes alone provide a strong predictive baseline and the handcrafted feature block produces a smaller, model-dependent change. A graph-size-grouped stress test showed a pronounced decline, indicating sensitivity to repeated graph-size signatures across folds. The study supports LLVM-IR-derived CPG embeddings for within-dataset vulnerability classification, while generalization to unseen crates and security advisories requires independent evaluation.

Keywords: Rust, software vulnerability detection, Code Property Graph, LLVM intermediate representation, Weisfeiler–Lehman, XGBoost

1 Introduction

Rust combines systems-level control with compile-time ownership and borrowing checks that prevent many memory-management errors [1, 2]. These guarantees narrow the attack surface, but they do not eliminate defects introduced through

unsafe code, foreign-function interfaces, dependency behavior, incorrect validation, concurrency logic, or protocol implementation. The ongoing expansion of the RustSec Advisory Database validates the necessity for analytical techniques that enhance the compiler [3].

Learned vulnerability detectors commonly represent programs as tokens, slices, or function-level graphs. Token models retain lexical context but may also exploit naming conventions and dataset-specific biases. Intermediate representations expose operations and control transfers in a more normalized form, while Code Property Graphs (CPGs) make structural and semantic relations directly available to graph analysis [4, 5]. The central question in this work is whether the transition from Rust source to LLVM IR and then to a CPG preserves security-relevant information that can be summarized compactly and learned from a modest dataset.

The experiment begins with advisory-guided Rust source collection, compiles the retained source files to LLVM IR, and converts the resulting IR into CPG JSON graphs with the Fraunhofer AISEC framework [6]. The same processing workflow continues through graph validation, feature construction, and model evaluation, allowing the computational procedure to be rerun from source collection to the reported results. Each graph is transformed into a 182-dimensional representation comprising 132 pooled node attributes, 30 projected Weisfeiler–Lehman (WL) components, and 20 handcrafted graph descriptors. The 182-dimensional representation is assessed with three conventional classifiers—XGBoost, Random Forest, and RBF-SVM—while a graph convolutional network (GCN) serves as a separate end-to-end baseline that learns directly from the CPG structure. Feature-component comparisons measure the contribution of node pooling, WL neighborhoods, and global structural statistics.

Four focused research questions organize the study:

RQ1 Does a compact, explicit 182-dimensional summary of an LLVM-IR-derived CPG retain sufficient class-discriminative signal to distinguish safe-labeled from vulnerable-labeled Rust CPGs?

RQ2 Among pooled node attributes, WL-refined local topology, and handcrafted global descriptors, which component drives predictive performance, and are the other components merely redundant or genuinely complementary?

RQ3 How do gradient-boosted and bagged tree ensembles and an RBF-SVM operating on the 182-dimensional representation compare with a graph-convolutional network trained directly on the CPG structure at the available sample size?

RQ4 How sensitive is the observed performance to the distribution of repeated graph-size signatures across training and test folds, and what does that sensitivity imply for interpretation of the within-dataset results?

The paper makes the following contributions:

1. It implements and documents an end-to-end RustSec-to-Rust-source to LLVM-IR-to-CPG workflow covering advisory-directed retrieval, source filtering, LLVM IR generation, CPG conversion, JSON validation, and class-specific exclusion accounting, allowing the dataset-construction process to be rerun from the original collection stage.
2. It develops and evaluates a 182-dimensional hybrid embedding that integrates node-level pooled statistics, two-hop WL neighborhood refinement, and global graph descriptors, and uses feature-component comparisons to determine how each block contributes to predictive performance.
3. It compares XGBoost, Random Forest, and RBF-SVM using the fixed 182-dimensional representation with a GCN trained directly on the graph structure under the same five-fold partitions, providing a comparison between explicit graph embeddings and end-to-end graph learning at the available sample size.
4. It supplements overall accuracy with balanced accuracy, MCC, ROC-AUC, PR-AUC, class-specific recall, Brier score, paired model analysis, and a conservative split-sensitivity test, quantifying how the reported performance changes when repeated graph-size signatures are constrained across folds.
5. It positions the CPG representation against established source-token, source-graph, and lexical-pretraining vulnerability detectors through a qualitative comparison of representation choice, learning method, and prediction unit, clarifying the design trade-off addressed by this study without claiming empirical superiority over methods evaluated on different datasets.

The scope is intentionally precise. The current evidence concerns classification within the retained CPG dataset. It provides no evidence of line-level localization, causal attribution to a disclosed flaw, or generalization to previously unseen crates and advisories.

2 Background and Related Work

2.1 Rust security beyond the type system

RustBelt formalized the reasoning principles that connect unsafe implementations with safe public abstractions [2]. That work clarifies why Rust’s safety mechanism depends not only on the compiler but also on the correctness of libraries that encapsulate unsafe behavior. RustSec provides the practical disclosure layer used in this study, identifying affected crates, vulnerable version ranges, repaired releases, and public references [3]. Static analyses such as SafeDrop further demonstrate that compiler-assisted reasoning can uncover Rust-specific memory-management defects that remain possible in real Rust programs [7].

2.2 Source, LLVM-IR, and CPG representations

Source-based detectors preserve identifiers, statements, and local syntax. VulDeePecker showed that semantically related code gadgets can support learned vulnerability recognition [8]; LineVul later used a pretrained transformer for function-level prediction and line ranking [9]. These approaches benefit from lexical context, but their outputs can be influenced by naming conventions, duplicate code, and project-specific regularities.

LLVM IR provides a lower-level, typed representation in which instructions and control transfers are explicit [4]. Rust-IR-BERT demonstrated that textual Rust-generated LLVM IR can be paired with a pretrained code transformer and a downstream classifier [10]. The present study moves one step further by converting LLVM IR into a graph and separating local neighborhood structure from global graph statistics.

CPGs integrate program entities and semantic relations in a unified graph [5]. The construction overlays several classically separate program

representations, an abstract syntax tree, a control-flow or execution-order graph, and one or more dependency graphs, onto a shared node set, so that syntactic, control-flow, and data-flow relations become simultaneously traversable edges, eliminating the need to analyze these representations separately. Devign established the relevance of comprehensive program graphs for neural vulnerability classification [11], and IVDetect showed how program dependency graphs can support fine-grained vulnerability interpretation [12]. ReVeal’s evaluation of deep vulnerability detectors is equally important: it showed that duplicates, sampling choices, and unrealistic partitions can produce optimistic estimates [13]. Those observations motivate the explicit exclusion audit and split-sensitivity analysis reported here. The original CPG formulation and the majority of its vulnerability-detection descendants operate on source or near-source graphs extracted from C or C++. The present study instead builds the CPG from compiler-lowered LLVM IR of Rust programs, so node and edge semantics reflect instruction-level operations, typed memory access, and explicit control transfers, not surface-level source syntax; this shifts the representation one level closer to the executable program while retaining the multi-relational structure that made the original CPG formulation effective for vulnerability discovery.

2.3 WL neighborhoods and compact graph learning

The Weisfeiler–Lehman procedure iteratively relabels each node from its current state and the multiset of neighboring labels, producing graph signatures that encode progressively larger local neighborhoods [14]. This relabeling is the color-refinement routine underlying the 1-WL graph isomorphism test: after h iterations, node labels reflect increasingly large subtree-like neighborhood patterns, so the resulting histogram provides a principled, bounded structural fingerprint. The computational cost of H refinement rounds scales linearly with the number of graph edges and the number of iterations, making WL signatures suitable for large graphs without requiring end-to-end message passing. The known limitation is that 1-WL cannot distinguish every pair of non-isomorphic graphs and, used alone, does not directly retain continuous node-level attributes

beyond the discrete labels used for refinement or the global graph statistics captured by the handcrafted block. WL features are therefore useful here as a complementary channel: they encode local topology without requiring a large end-to-end neural model, at the cost of a coarser representation of node attributes and no explicit long-range structure beyond the chosen number of iterations. The hybrid representation combines these signatures with pooled node attributes and handcrafted global descriptors so that local topology, node-attribute statistics, and whole-graph structural characteristics are represented separately. XGBoost and Random Forest then learn nonlinear interactions among these components, while the GCN tests whether direct, trainable message passing offers an advantage over the fixed feature representation at the available sample size.

2.4 Qualitative positioning against established studies

Table 1 compares the representation techniques and scientific scope of highly relevant studies. The table deliberately avoids ranking methods by published accuracy because the datasets, languages, labeling units, and validation protocols are not interchangeable.

3 Dataset Construction and JSON File Audit

3.1 Advisory-guided collection

The dataset was generated through a staged workflow beginning with RustSec security advisories. Advisory metadata was used to identify affected and repaired crate versions, retrieve the corresponding Rust packages, and retain relevant Rust source files according to the study’s filtering rules. Each retained source file was compiled to textual LLVM IR, and the resulting IR was converted to a JSON CPG with the Fraunhofer AISEC framework. Each experimental sample is one CPG generated from the compiled LLVM IR of a single retained Rust source file. Figure 1 summarizes the dataset-generation workflow.

3.2 Label Construction and Validity

RustSec advisories were used to identify affected and repaired crate versions. Source files retained from affected versions were assigned the vulnerable-labeled class, whereas files retained from repaired versions were assigned the safe-labeled class. These labels were carried forward through LLVM IR generation and CPG construction. The labels therefore reflect the advisory status of the corresponding crate version and are not based on file-level localization of the reported vulnerability. A source file from an affected version may not itself contain the advisory-related code, and a crate version may contribute multiple retained source files. Likewise, files from repaired versions are described as safe-labeled because the study does not independently verify that each file is free of all vulnerabilities. The labels are therefore used as dataset class labels for the within-dataset classification task, not as file-level proof of vulnerability presence or absence.

3.3 Rust source to LLVM IR

Rust source files were compiled to textual LLVM IR as part of the pre-processing workflow. This intermediate form exposes low-level instructions, calls, branches, types, and memory operations in a compiler-oriented representation. LLVM IR reduces source-level syntactic variance, but its exact form remains dependent on compiler version, compilation flags, optimization level, crate configuration, and target architecture. These settings form part of the reproducibility boundary, and the compilation stage can be rerun from the retained Rust source files using the documented processing procedure.

3.4 LLVM IR to CPG

The LLVM files were translated into JSON CPGs with the Fraunhofer AISEC CPG framework [6]. This conversion is part of the automated data-generation workflow and produces the graph files used by the downstream experiment. The exported graphs contain typed nodes, properties, and semantic edges. The experiment recognizes ten relation types: AST, DFG, EOG, PDG, CDG, BB, USAGE, REFERS_TO, INVOKES, and SCOPE. WL refinement uses the AST, DFG, PDG, CDG, and EOG relations.

Table 1 Qualitative positioning against representative vulnerability-detection studies.

Study	Representation	Learning method	Prediction unit	Relevance to this study
VulDeePecker [8]	Data-flow-oriented source gadgets	Recurrent neural network	Code gadget	Established learned detection over semantically related source fragments, without an explicit whole-program CPG.
SySeVR [15]	Syntax- and semantics-based source-code slices	Bidirectional RNN/GRU network	Code slice	Extends slice-based source detection with syntax- and semantics-aware candidate extraction, but likewise stops short of a whole-program CPG.
Devign [11]	Rich syntax and dependency graph	Gated graph neural network	Function	Representative graph-neural vulnerability classifier and an important reference for graph-based vulnerability detection.
ReVeal analysis [13]	Source and graph-informed representations	Multiple deep models	Function	Demonstrated the importance of duplicate control, realistic sampling, and split design.
IVDetect [12]	Program dependency graph	Interpretable graph model	Function and statement	Adds fine-grained interpretation; the present work remains graph-level without localizing vulnerable lines.
LineVul [9]	Source-code tokens	Pretrained transformer	Function and line	Uses lexical pretraining and attention for function- and line-level prediction, providing a source-token comparison with the explicit LLVM-IR-derived graph representation evaluated here.
Rust-IR-BERT [10]	Textual Rust-generated LLVM IR	GraphCode BERT with CatBoost	LLVM-IR sample	Directly relevant prior Rust vulnerability-detection work operating on textual LLVM IR before conversion to the CPG representation used here.
This study	LLVM-IR-derived CPG with pooled, WL, and graph-descriptor components	XGBoost, Random Forest, RBF-SVM, and GCN	Whole CPG	Examines a compact structural representation, feature-component contributions, class-specific metrics, and split sensitivity.

3.5 JSON-stage filtering and class-specific attrition

After source collection, LLVM IR generation, and CPG conversion, 256 JSON CPG files entered the validation and filtering stage: 161 safe-labeled and 95 vulnerable-labeled. The parser identified four malformed vulnerable files. Three safe graphs contained fewer than ten nodes and were excluded by the minimum-size rule. Fourteen graphs exceeded the predefined 50,000-node processing limit: thirteen safe and one vulnerable. No byte-identical duplicate was found by SHA-256 hashing.

The final dataset therefore contains 235 graphs: 145 safe-labeled and 90 vulnerable-labeled. Table 2 reports every JSON-stage exclusion. The unequal effect of the large-graph filter is treated as a potential concern for representativeness.

Table 2 JSON-stage data accounting and exclusions. Safe and vulnerable denote the assigned class labels.

Status	Safe	Vulnerable	Total
Generated JSON CPGs	161	95	256
Malformed JSON	0	4	4
Above 50,000 nodes	13	1	14
Below 10 nodes	3	0	3
Exact duplicates removed	0	0	0
Final usable dataset	145	90	235

3.6 Graph-size characteristics

Safe graphs contain a mean of 7,860.0 nodes (standard deviation 12,230.9) and 43,722.8 edges (standard deviation 69,921.9). Vulnerable graphs are larger on average, with 9,952.4 nodes (standard

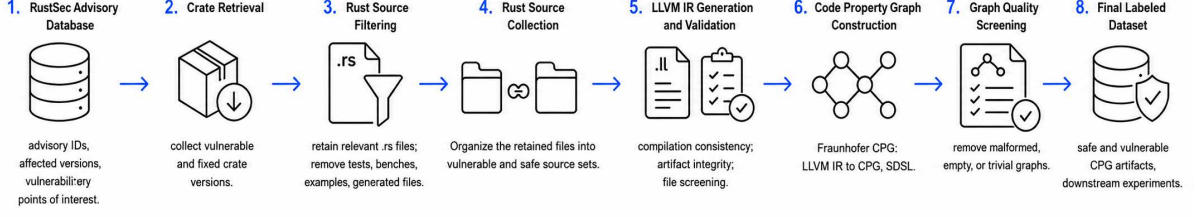


Fig. 1 Dataset collection pipeline for constructing the labeled Rust CPG dataset

deviation 10,701.3) and 56,125.8 edges (standard deviation 60,996.1). Both classes include graphs with 599 nodes at the lower bound. The retained maxima are 49,634 nodes for safe graphs and 45,353 nodes for vulnerable graphs. Table 3 and Figure 2 summarize these characteristics.

Thirty-two groups share an identical $(|V|, |E|)$ signature. Such equality cannot establish code duplication, but it indicates repeated graph-size patterns and motivates the sensitivity analysis in Section 5.5.

4 CPG Representation and Learning Pipeline

4.1 Overview

Let a CPG be

$$G = (V, E, \tau_V, \tau_E, X), \quad (1)$$

where V denotes the node set, E the set of directed semantic edges, τ_V node types, τ_E edge types, and X node attributes. Because G is typed and multi-relational, no single summary statistic is expected to capture all potentially security-relevant structure: node-level attributes describe what each program entity is, WL-refined neighborhoods describe how nearby entities are typed and connected, and whole-graph descriptors describe aggregate program shape. The representation pipeline is organized around this three-level decomposition, and the three resulting blocks are concatenated into a single fixed-length vector before classification. The traditional classifiers use this fixed-dimensional embedding, while the GCN directly processes node

attributes and a bidirectional edge index, learns neighborhood representations through trainable graph convolutions, and then applies global mean and maximum pooling. It therefore does not use the fixed WL signature or the precomputed pooled-feature block. Figure 3 summarizes the end-to-end experimental pipeline.

4.2 Node attributes

The dataset contains 44 observed node types. Ten semantic edge categories are represented by separate incoming and outgoing degree features. Each node receives a 66-dimensional vector:

$$\begin{aligned} F &= C + 2K + 2 \\ &= 44 + 2(10) + 2 = 66, \end{aligned} \quad (2)$$

where C is the one-hot node-type dimension, K is the number of semantic edge categories, and the final two values are total incoming and outgoing degree. Degree values are transformed with $\log(1 + x)$.

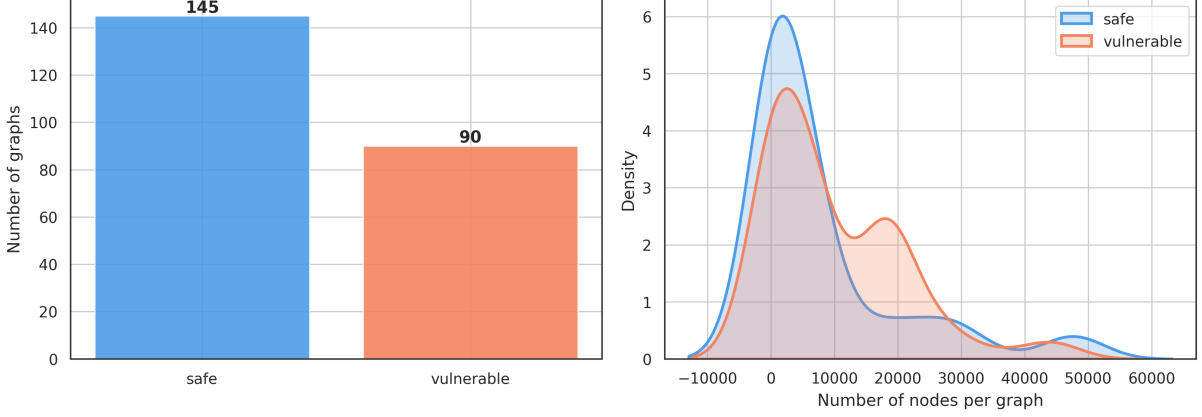
The node-type vocabulary is derived from the full dataset. It excludes labels and class-dependent statistics, but it remains a mildly transductive schema choice. For the fixed-feature classifiers, continuous node scaling and WL projection are fitted within each training fold.

4.3 Weisfeiler–Lehman signatures

The WL procedure refines a node label from its previous label and a sorted multiset of neighboring

Table 3 Descriptive graph statistics after filtering.

Class	Quantity	Mean	Std. dev.	Minimum	Maximum
Safe	Nodes	7,860.0	12,230.9	599	49,634
Safe	Edges	43,722.8	69,921.9	2,414	272,869
Vulnerable	Nodes	9,952.4	10,701.3	599	45,353
Vulnerable	Edges	56,125.8	60,996.1	2,414	248,758

**Fig. 2** Class distribution and graph-size overview for the 235 retained CPGs

labels [14]. For node v at iteration h ,

$$\ell_v^{(h)} = \text{hash} \left(\ell_v^{(h-1)} \parallel \text{sort} \left\{ \ell_u^{(h-1)} : u \in \mathcal{N}^+(v) \right\} \right). \quad (3)$$

Directed edges from the AST, DFG, PDG, CDG, and EOG relation sets are followed through $\mathcal{N}^+(v)$, making the refinement sensitive to edge direction while aggregating the selected relations.

The implementation records iteration 0 and performs two refinement rounds, so the resulting labels summarize each node’s two-hop neighborhood over the selected directed relations. To bound processing for very high-degree nodes, at most 200 neighbor labels are retained through deterministic sampling; this trades some neighborhood fidelity at high-degree nodes for bounded per-node processing cost. The per-graph label histogram is, in principle, the explicit feature map whose inner product defines the WL subtree kernel [14]; because the WL label vocabulary can be large and is not known a priori, graph-level WL label counts are mapped into 4,096 signed hash bins (the sign chosen by a second independent hash of the label) and normalized by the absolute count sum, which keeps the expected contribution of colliding labels near zero while avoiding systematically inflating

any bin. A training-fold Truncated SVD projection then reduces this hashed block to 30 dimensions; therefore, the SVD basis is learned without using test-fold graphs. Feature hashing controls memory consumption and avoids learning an explicit dataset-wide WL vocabulary, at the cost of an approximation whose fidelity depends on the bin count relative to the number of distinct labels actually observed. Figure 4 illustrates a representative local neighborhood, and Figure 5 depicts the two-stage refinement process.

4.4 Pooled node component

Within each fold, node attributes are standardized using the training graphs. Mean and maximum pooling summarize the resulting node matrix:

$$\mathbf{z}_G^{\text{pool}} = \left[\frac{1}{|V|} \sum_{v \in V} \tilde{\mathbf{x}}_v \parallel \max_{v \in V} \tilde{\mathbf{x}}_v \right]. \quad (4)$$

With 66 node attributes, concatenated mean and maximum pooling contributes 132 dimensions. Both operators are permutation-invariant in V , which is appropriate because a CPG carries no canonical node ordering; mean pooling summarizes the typical node profile of the graph, while

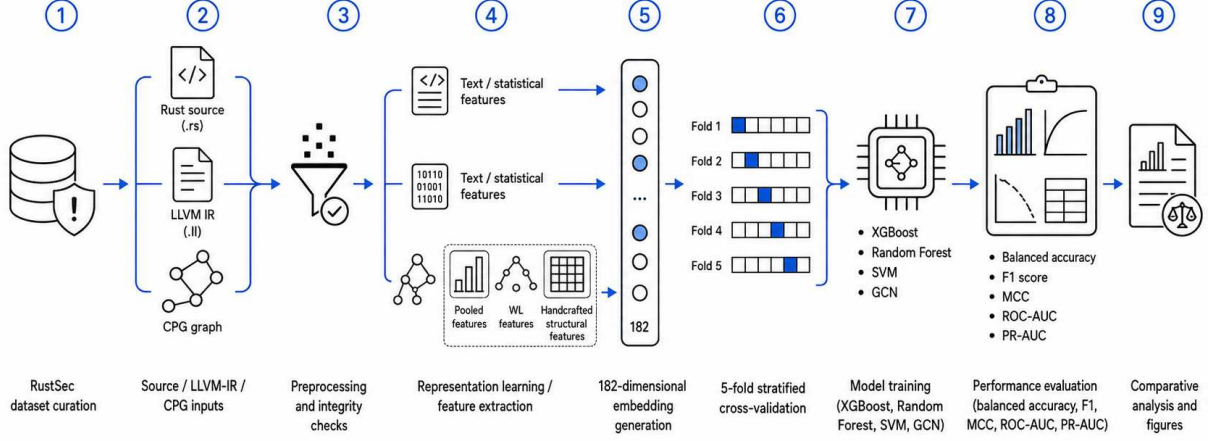


Fig. 3 Overall experimental pipeline for Rust vulnerability classification using Rust source, LLVM IR, and CPG representations, followed by feature extraction, 182-dimensional embedding, five-fold cross-validation, model training, and performance evaluation

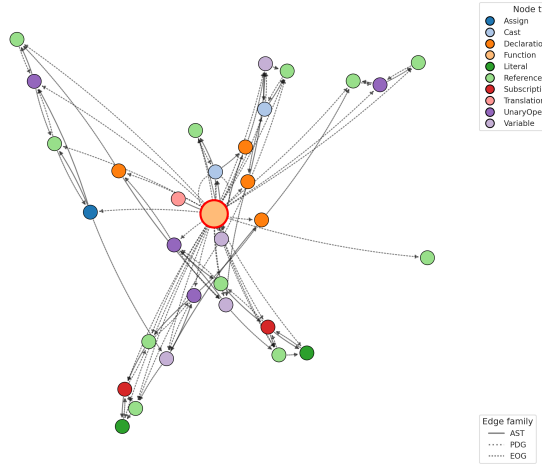


Fig. 4 Representative two-hop CPG neighborhood centered on a function node. Nodes are grouped by CPG type and edge styles denote semantic relations

maximum pooling preserves sensitivity to rare but extreme attribute values, such as a single node with unusually high semantic-edge degree, that averaging alone would smooth away.

4.5 Handcrafted graph descriptors

The 20 features in this block are logarithmically transformed counts or summaries of:

- graph nodes and edges;
- functions, basic blocks, variables, calls, casts, and unary operators;
- DFG, CDG, and PDG edges;
- pointer dereferences, address-of operations, pointer casts, and global variables;
- aggregate function complexity;
- approximate LLVM opcode frequencies for `alloca`, `getelementptr`, `invoke`, and `store`.

Opcode frequencies are extracted heuristically from function-node code strings. They are approximate indicators, not a replacement for parsing LLVM IR with a dedicated instruction parser.

4.6 Hybrid embedding

The complete vector is

$$\mathbf{z}_G = \mathbf{z}_G^{\text{pool}} \parallel \mathbf{z}_G^{\text{WL}} \parallel \mathbf{z}_G^{\text{HC}}. \quad (5)$$

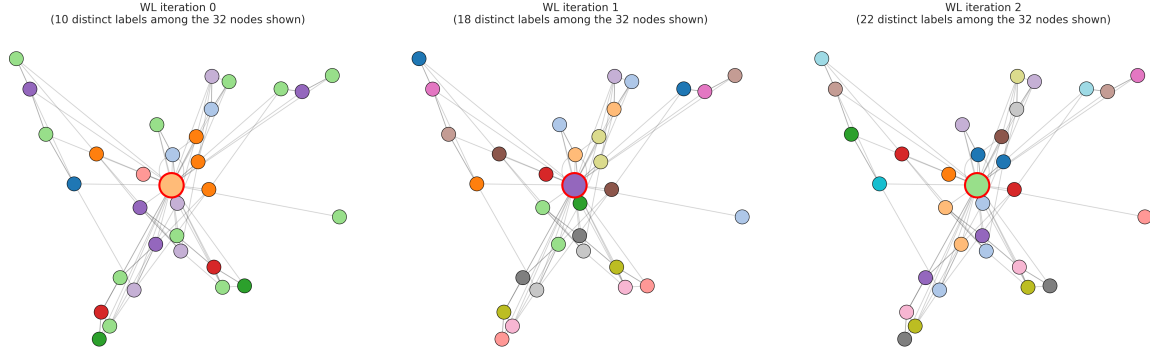


Fig. 5 Two-stage WL refinement. Iteration 0 uses node types; iterations 1 and 2 encode progressively larger local neighborhoods

The resulting dimensionality is

$$\begin{aligned} \dim(\mathbf{z}_G) &= 132 + 30 + 20 \\ &= 182. \end{aligned} \quad (6)$$

The embedding transformer is implemented within the scikit-learn pipeline [16]. Node scaling and WL SVD are therefore estimated from each training fold distinct from the corresponding test fold.

4.7 Classifiers

4.7.1 XGBoost

The primary tree-boosting model uses 200 estimators, maximum depth 4, learning rate 0.05, and a positive-class weight equal to the ratio of safe-labeled to vulnerable-labeled samples in the retained dataset [17].

4.7.2 Random Forest

The Random Forest contains 300 trees, balanced class weights, and a fixed random seed [18].

4.7.3 RBF-SVM

The SVM uses the radial basis function kernel with $C = 1$, automatic γ , balanced class weights, and probability estimation.

4.7.4 GCN

The GCN contains two graph convolutional layers with 64 hidden units, ReLU activation, 0.3 dropout, global mean and maximum pooling, and

a final two-class linear layer [19]. It is trained for 100 epochs per fold with Adam, learning rate 10^{-3} , weight decay 10^{-4} , and class-weighted cross-entropy. PyTorch Geometric is used for graph batching and message passing [20, 21]. The GCN uses a bidirectional edge index without relation-specific message-passing parameters. It therefore serves as a homogeneous graph-neural baseline; edge-aware and heterogeneous message passing are outside the scope of this baseline.

5 Experimental Design

5.1 Cross-validation

The primary experiment uses five-fold stratified cross-validation with shuffling and a fixed seed. Every graph receives one out-of-fold probability and label from each model. The same dataset and outer partitioning logic are used across the four classifiers, although the GCN is trained through a separate fold loop.

The protocol estimates performance on held-out graphs from the same dataset. Crate-disjoint, advisory-disjoint, and temporal partitions were not used in the current evaluation.

5.2 Metrics

Accuracy is supplemented with balanced accuracy, macro-F1, Matthews correlation coefficient (MCC), ROC-AUC, PR-AUC, vulnerable-class precision and recall, safe-class specificity, and Brier score. Balanced accuracy and macro-F1 assign equal importance to the two classes. MCC summarizes the full confusion matrix and remains informative

under class imbalance [22]. PR–AUC emphasizes vulnerable-class retrieval and is more revealing than ROC–AUC when the positive class is less prevalent [23]. Brier score measures probabilistic prediction error.

Fold-level 95% confidence intervals are computed with a Student- t interval. Class-stratified bootstrap intervals for the pooled out-of-fold metrics are also generated and stored with the reproducibility outputs.

5.3 Feature-component analysis

XGBoost and Random Forest are trained with seven feature configurations:

1. pooled node attributes only;
2. WL component only;
3. handcrafted descriptors only;
4. pooled attributes plus WL;
5. pooled attributes plus handcrafted descriptors;
6. WL plus handcrafted descriptors;
7. complete Pool+WL+HC representation.

5.4 Pairwise comparison

The leading two models are compared by a paired bootstrap of the ROC–AUC difference. The test operates on matched out-of-fold predictions, avoiding an unpaired comparison of independently averaged scores.

5.5 Graph-size signature sensitivity

A secondary stress test uses $(|V|, |E|)$ as a grouping key in five-fold StratifiedGroupKFold. Graphs with the same node and edge counts cannot appear on opposite sides of a split. This signature is not a verified crate, advisory, clone, or code-family identifier. The analysis is consequently interpreted as a split-sensitivity test.

6 Results

6.1 Overall classifier performance

Table 4 reports the complete out-of-fold metric suite. XGBoost obtains the strongest observed balance between vulnerable-labeled recall and safe-labeled specificity. Random Forest remains close in ranking performance, whereas the GCN shows substantially higher safe-labeled specificity than vulnerable-labeled recall.

Figure 6 plots the five-fold comparison across classifiers. The XGBoost confusion matrix contains 119 correctly classified safe-labeled graphs, 26 false positives, 22 false negatives, and 68 correctly classified vulnerable-labeled graphs (Figure 7). Forty-eight of 235 CPGs are misclassified (20.4%). The error rate is 17.93% for safe graphs and 24.44% for vulnerable graphs. Figure 8 reports the corresponding out-of-fold ROC curves.

6.2 Fold-level estimates and uncertainty

Table 5 reports fold means with descriptive Student- t 95% interval half-widths. XGBoost reaches a mean ROC–AUC of 0.9086 ± 0.0305 . Random Forest follows closely at 0.8969 ± 0.0195 . Table 4 computes metrics from all pooled out-of-fold predictions, whereas Table 5 reports the arithmetic mean of metrics calculated separately within each fold. Nonlinear metrics such as ROC–AUC and macro-F1 therefore differ slightly between the two summaries. The paired bootstrap estimates the XGBoost–Random Forest AUC difference as 0.0128, with 95% interval $[-0.0070, 0.0345]$. Since the interval includes zero, the data do not establish a statistically reliable superiority of XGBoost over Random Forest.

6.3 Feature importance and class-level differences

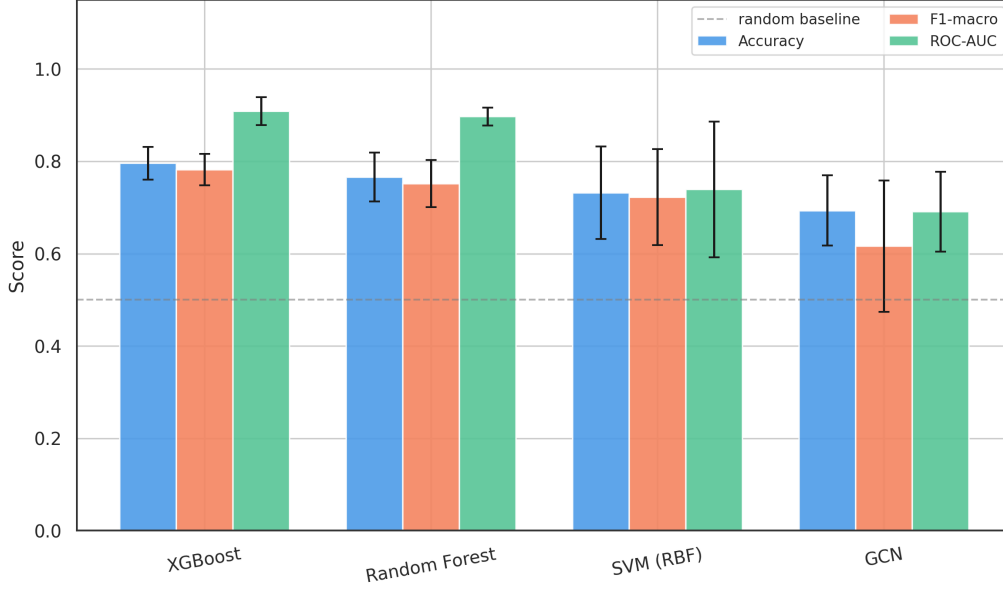
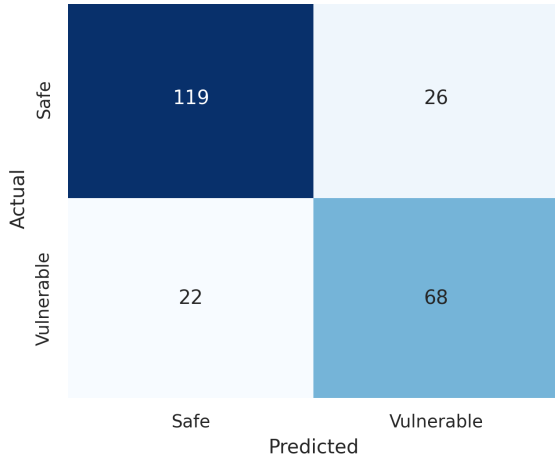
The tree-based importance analysis is descriptive because the displayed model is fitted to the full dataset after evaluation. It is used to describe which features receive greater importance in the fitted model, not to estimate predictive performance. Figure 9 presents the leading Random Forest feature importances. Figure 10 contrasts mean engineered-feature values between safe-labeled and vulnerable-labeled graphs.

6.4 Feature-component findings

Table 6 reports all seven representation settings. For XGBoost, the complete embedding obtains the highest ROC–AUC (0.9086). Pool+WL follows at 0.9048; adding the 20 engineered graph descriptors to Pool+WL produces a small increase in ROC–AUC to 0.9086. Pooling alone remains comparatively strong at 0.8914, while WL-only and handcrafted-only models are weaker.

Table 4 Out-of-fold results on 235 CPGs. The vulnerable-labeled class is treated as positive.

Model	Acc.	Bal. Acc.	Prec.	Recall	Spec.	Macro-F1	MCC	ROC-AUC	PR-AUC	Brier
XGBoost	0.7957	0.7881	0.7234	0.7556	0.8207	0.7856	0.5718	0.9033	0.8603	0.1220
Random Forest	0.7660	0.7556	0.6882	0.7111	0.8000	0.7539	0.5081	0.8903	0.8573	0.1304
RBF-SVM	0.7319	0.7301	0.6311	0.7222	0.7379	0.7231	0.4508	0.6935	0.5493	0.2061
GCN	0.6936	0.6400	0.6607	0.4111	0.8690	0.6423	0.3196	0.6790	0.6232	0.2094

**Fig. 6** Five-fold classifier comparison with fold-level 95% confidence intervals**Fig. 7** Out-of-fold confusion matrix for XGBoost

Random Forest behaves differently. Pool+WL reaches ROC-AUC 0.8981, marginally exceeding the complete representation at 0.8969. The result cautions against claiming that every component is

universally beneficial. Rather, the relative contribution of each feature component varies across classifiers. Figure 11 visualizes the feature-component results for both classifiers.

6.5 Split-sensitivity result

Table 7 contrasts the principal stratified split with the node/edge-signature stress test. Performance falls sharply when all graphs sharing the same size signature are kept together. In this grouped split, balanced accuracy and ROC-AUC fall below 0.5 and MCC becomes negative.

The stress test should not be described as crate-disjoint validation. Graphs with equal node and edge counts may be unrelated, and the grouping can create an artificial distribution shift. The decline shows that performance is sensitive to how repeated size signatures are distributed across folds; the stress test should therefore be interpreted as a robustness check within the dataset, not as evidence of cross-project generalization or failure.

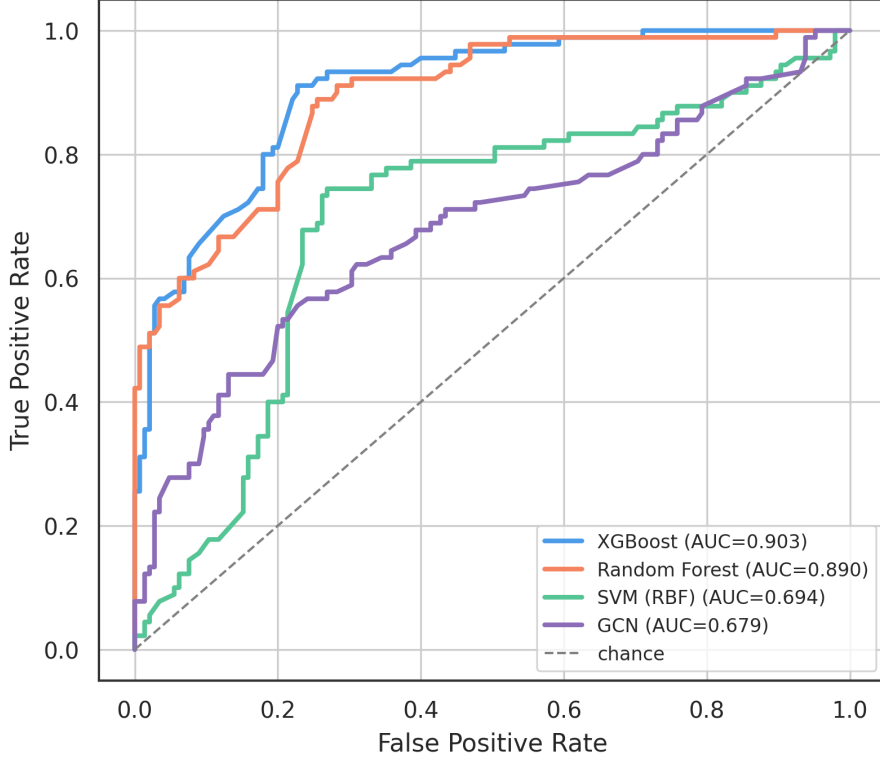


Fig. 8 Out-of-fold ROC curves for XGBoost, Random Forest, RBF-SVM, and GCN

Table 5 Five-fold means and descriptive Student-*t* 95% interval half-widths.

Model	Accuracy	Macro-F1	ROC-AUC
XGBoost	0.7957 ± 0.0354	0.7824 ± 0.0341	0.9086 ± 0.0305
Random Forest	0.7660 ± 0.0528	0.7518 ± 0.0513	0.8969 ± 0.0195
RBF-SVM	0.7319 ± 0.0999	0.7225 ± 0.1037	0.7393 ± 0.1471
GCN	0.6936 ± 0.0761	0.6162 ± 0.1424	0.6908 ± 0.0866

7 Discussion

7.1 RQ1: within-dataset vulnerability discrimination

The hybrid CPG representation contains substantial class-discriminative information under ordinary five-fold stratification. XGBoost reaches 78.81% balanced accuracy and ROC-AUC 0.9033, with vulnerable recall 75.56%. These results demonstrate within-dataset ranking and classification performance under the stated protocol. They provide no basis for assuming the same performance on a new crate family, a future advisory, or a separately generated CPG dataset.

7.2 RQ2: contribution of graph components

Pooling accounts for a large share of the predictive signal. This is plausible because mean and maximum aggregation summarize node-type and semantic-degree profiles across each graph. Adding the WL component increases the observed XGBoost ROC-AUC from 0.8914 to 0.9048, suggesting complementary information from local topology. The manually designed block produces a smaller observed increase for XGBoost and no increase for Random Forest beyond Pool+WL. The evidence supports a combined representation, but not the stronger assertion that every component is indispensable.

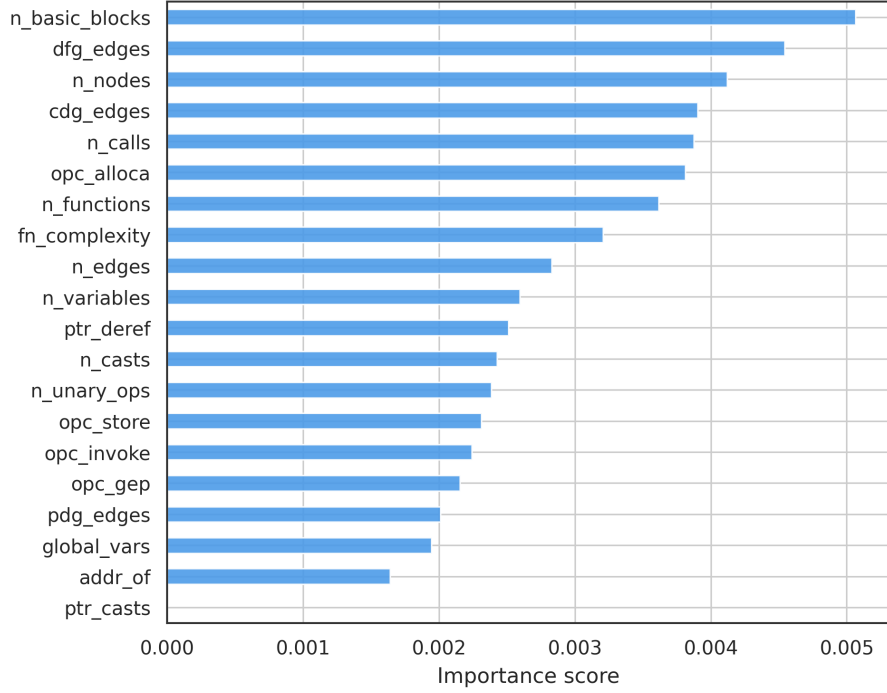


Fig. 9 Random Forest feature importance from the model fitted to the complete dataset for interpretation only

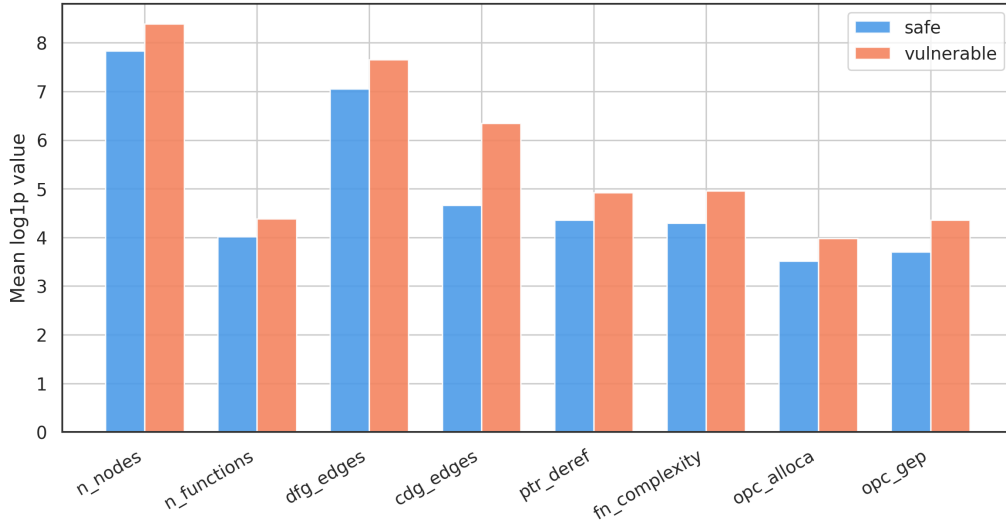


Fig. 10 Mean values of selected predefined graph descriptors for safe and vulnerable CPGs. Values are descriptive and should not be read as causal vulnerability indicators

7.3 RQ3: comparison of model families

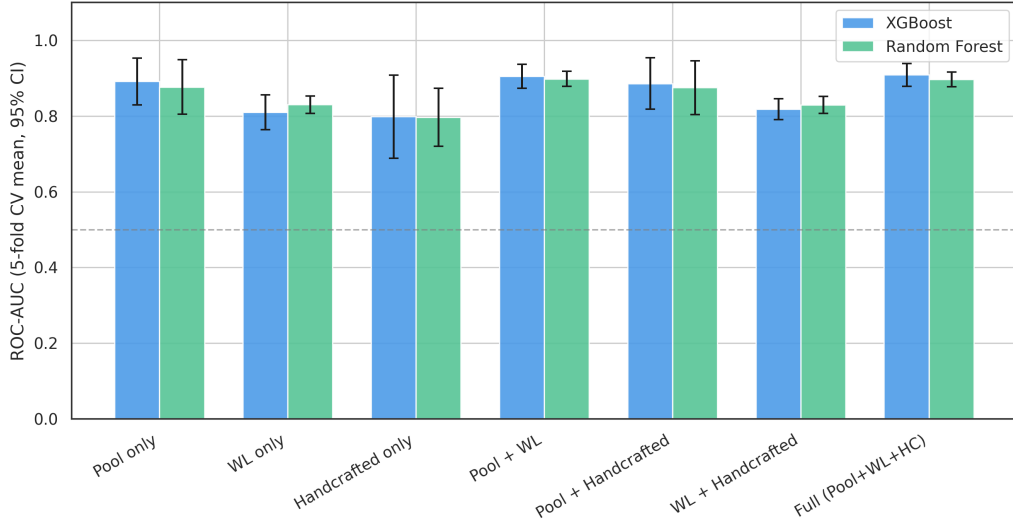
XGBoost and Random Forest achieve higher observed scores than the RBF-SVM and GCN. XGBoost obtained the highest observed score, but

the paired AUC interval includes zero and therefore provides insufficient evidence of statistical superiority over Random Forest.

The weaker GCN should not be interpreted as evidence that graph neural networks are unsuitable for vulnerability analysis. Devign and IVDetect

Table 6 Feature-component comparison under five-fold cross-validation. CI columns are 95% half-widths.

Classifier	Embedding	ROC-AUC	AUC CI	Macro-F1	F1 CI	Accuracy
XGBoost	Pool only	0.8914	0.0619	0.7777	0.1005	0.7830
	WL only	0.8100	0.0461	0.7050	0.0420	0.7234
	Handcrafted only	0.7985	0.1100	0.7158	0.1272	0.7277
	Pool + WL	0.9048	0.0319	0.7650	0.0418	0.7787
	Pool + handcrafted	0.8856	0.0678	0.7737	0.1010	0.7787
	WL + handcrafted	0.8180	0.0279	0.7050	0.0420	0.7234
	Full Pool+WL+HC	0.9086	0.0305	0.7824	0.0341	0.7957
Random Forest	Pool only	0.8768	0.0722	0.7567	0.0990	0.7617
	WL only	0.8299	0.0231	0.7128	0.0552	0.7319
	Handcrafted only	0.7969	0.0764	0.6975	0.1178	0.7106
	Pool + WL	0.8981	0.0199	0.7600	0.0549	0.7745
	Pool + handcrafted	0.8749	0.0706	0.7567	0.0990	0.7617
	WL + handcrafted	0.8295	0.0228	0.7049	0.0542	0.7234
	Full Pool+WL+HC	0.8969	0.0195	0.7518	0.0513	0.7660

**Fig. 11** Contribution of pooled node attributes, WL signatures, and engineered graph descriptors for XGBoost and Random Forest

demonstrate the capacity of graph models on larger, carefully constructed datasets [11, 12]. In this experiment, the GCN is trained from scratch on only 235 heterogeneous graphs. The GCN achieves substantially lower vulnerable-labeled recall at the available sample size.

7.4 RQ4: robustness and split dependence

The dataset review identifies two issues relevant to robustness. First, the graph-size limit removes sixteen safe graphs but only one vulnerable graph when the three undersized graphs are also considered. Second, 32 repeated node/edge-count signatures remain.

The grouped sensitivity result makes this limitation concrete. The grouping key is deliberately coarse, so the stress test cannot establish that ordinary stratified cross-validation is invalid. It does show that the observed performance is sensitive to how repeated size signatures are distributed across folds.

7.5 Relationship to established vulnerability detectors

The present model differs from the source-based and graph-neural systems summarized in Table 1. VulDeePecker [8] and SySeVR [15] operate on selected source-code slices with sequence models; Devign and IVDetect operate on source-derived

Table 7 XGBoost sensitivity to the partitioning rule.

Protocol	Acc.	Bal. Acc.	Macro-F1	MCC	ROC-AUC	PR-AUC
Ordinary stratified CV	0.7957	0.7881	0.7856	0.5718	0.9033	0.8603
Grouped by (V , E)	0.4553	0.4406	0.4399	-0.1160	0.4146	0.4072

program graphs with end-to-end graph-neural learning, allowing the network to learn structural patterns directly; LineVul operates on source tokens with a pretrained transformer and large-scale lexical pretraining. The present approach builds an explicit, low-dimensional graph summary from a CPG derived from compiler-lowered LLVM IR and separates that summary into pooled node attributes, WL-refined local topology, and explicit graph-level descriptors. This combination explores a distinct representation-level design point between source-token methods and end-to-end source-graph models. It is computationally modest and interpretable at the feature-family level through the component analysis in Table 6, but it sacrifices statement-level localization and may map structurally different programs to similar graph summaries.

The comparison with Rust-IR-BERT is particularly instructive [10], since it is the closest prior work in both language target and representation level. Textual LLVM IR paired with a pretrained transformer can exploit instruction-level lexical and sequential context through pretrained contextual representations. The CPG approach exposes explicit structural relations, control- and data-dependency edges, typed nodes, and multi-hop neighborhoods as separately measurable components and supports the direct evaluation of the WL contribution reported in Section 5.3 and Table 6. Because the two studies use different datasets, sample sizes, and evaluation protocols, their reported metrics are not directly comparable; the contribution here is the representation-level component analysis, not a claim of superiority over the token-level approach.

7.6 Practical interpretation

A model trained under this protocol is best viewed as an experimental prioritization mechanism. It could rank CPGs for further inspection, but it should not automatically declare a crate secure or vulnerable. Practical use would require retraining and calibration on new releases, together with

integration with RustSec checks, static analyzers, tests, and manual review.

8 Reproducibility

The study was carried out through a documented computational workflow covering the full processing sequence from Rust source preparation to model evaluation. The workflow includes compiling retained `.rs` files to LLVM IR, converting the resulting LLVM IR into JSON Code Property Graphs (CPGs), validating and filtering the generated graphs, constructing graph representations, training the classifiers, and reproducing the reported evaluation results and figures.

The graph-generation toolchain used Rust 1.64.0, LLVM 14.0.6 with typed pointers, and Fraunhofer AISEC CPG 10.1.0. Rust source files were compiled to LLVM IR using:

```
rustup run 1.64.0 rustc --emit=llvm-ir -C
opt-level=0
-C codegen-units=1 <input.rs> -o <output.ll>
```

The corresponding compilation procedure is retained in `scripts/convert_to_ll.sh` in the replication package. LLVM IR files were converted to CPG JSON using the Fraunhofer AISEC `cpg-neo4j` command-line interface with the following configuration:

```
cpg-neo4j --export-json <output.json>
--no-neo4j
--no-default-passes
--custom-pass-list
"TypeHierarchyResolver,SymbolResolver,DFGPass,
EvaluationOrderGraphPass,TypeResolver,
ControlFlowSensitiveDFGPass,ProgramDependenceGraphPass"
<input.ll>
```

The exact safe and vulnerable class conversion procedures are retained in `scripts/safe_ll_to_CPG.sh` and `scripts/vulnerable_ll_to_CPG.sh`.

The classification experiments were executed with Python 3.12.13, NumPy 2.0.2, scikit-learn 1.6.1, XGBoost 3.3.0, and PyTorch 2.11.0+cpu. PyTorch Geometric was used for graph batching

and GCN message passing. A global random seed of 42 was applied to Python, NumPy, PyTorch, the classical estimators, the shuffled cross-validation procedure, and the WL Truncated SVD projection.

The primary evaluation used five-fold stratified cross-validation implemented with `StratifiedKFold(n_splits=5, shuffle=True, random_state=42)`. The same fold assignments were used for XGBoost, Random Forest, RBF-SVM, and the GCN.

Graph validation excluded malformed JSON files, graphs containing fewer than 10 nodes, and graphs containing more than 50,000 nodes. Exact duplicate files were checked using SHA-256 hashes. Starting from 256 CPG files (161 safe-labeled and 95 vulnerable-labeled), four malformed vulnerable graphs, thirteen oversized safe graphs, one oversized vulnerable graph, and three undersized safe graphs were excluded. No exact duplicate files were removed. The final dataset therefore contained 235 graphs: 145 safe-labeled and 90 vulnerable-labeled.

The experimental notebook uses fixed model configurations and does not perform automated hyperparameter optimization over the held-out folds. XGBoost used 200 estimators, maximum depth 4, learning rate 0.05, `eval_metric=logloss`, and a positive-class weight equal to the retained safe-to-vulnerable class ratio. Random Forest used 300 trees and balanced class weights. The RBF-SVM used $C = 1$, $\gamma = \text{scale}$, balanced class weights, and probability estimation. The GCN used two graph-convolutional layers with 64 hidden units, ReLU activation, dropout 0.3, global mean and maximum pooling, and a final two-class linear layer. It was trained for 100 epochs per fold using Adam with learning rate 10^{-3} , weight decay 10^{-4} , class-weighted cross-entropy, and batches of at most eight graphs.

For the graph-size-grouped sensitivity analysis, graphs sharing the same $(n_{\text{nodes}}, n_{\text{edges}})$ signature were kept in the same group. The resulting evaluation used five-fold `StratifiedGroupKFold` with shuffling and `random_state=119`. Confidence intervals for the final reported metrics were estimated using 2,000 stratified bootstrap resamples with random seed 2068. The paired XGBoost–Random Forest ROC–AUC comparison used 2,000 paired bootstrap resamples with random seed 42.

The replication package contains the source-processing and conversion scripts, generated CPG JSON files, PyTorch Geometric graph artifacts,

feature representations, dataset and exclusion manifests, model outputs, evaluation tables, figures, and the executed analysis notebook. Together, these materials record the principal software, preprocessing, partitioning, and model settings required to reproduce the reported experimental workflow.

9 Threats to Validity, Limitations, and Future Work

9.1 Internal Validity

Potential leakage was addressed by removing exact duplicates, excluding file paths and filenames from predictive features, and fitting continuous scaling and WL projection within each training fold. SHA-256 hashing detected no exact duplicates. Repeated node/edge-count signatures remain in the dataset; therefore, the grouped sensitivity analysis was included to examine the effect of partitioning on model performance.

9.2 Selection and Attrition Bias

The 50,000-node limit excludes 13 safe graphs and one vulnerable graph, while the minimum-size rule removes three additional safe graphs. These exclusions were imposed for memory stability, not predictive convenience, and were logged before model evaluation. Their class imbalance may still alter the estimated decision boundary. Results are conditional on the 235 retained graphs.

9.3 Limitations and Future Work

The principal limitations are the modest dataset size, class-skewed graph-size exclusions, file-level advisory labels, absence of an independent test set, and incomplete one-to-one traceability from the retained JSON CPG filenames back to the corresponding `.rs` and `.ll` files. The generation workflow itself remains reproducible, but the retained filenames provide limited retrospective linkage across these intermediate files. The model classifies whole CPGs without identifying vulnerable functions, statements, or paths. The duplicate audit identifies byte-identical CPG files and repeated graph-size signatures, but it does not establish semantic or source-level clone equivalence between affected and repaired samples.

A stronger follow-up should:

1. add a content-addressed manifest linking advisory, crate, version, source file, LLVM IR, and CPG for each generated sample;
2. evaluate crate-disjoint, advisory-disjoint, and temporal splits after establishing per-sample group identifiers;
3. eliminate or separately model large graphs without excluding them;
4. investigate heterogeneous and edge-aware GNNs after expanding the labeled graph dataset;
5. add statement-level explanations and patch-localized labels.

10 Conclusion

This study evaluates vulnerability classification from Code Property Graphs generated from Rust-produced LLVM IR. Its main technical contribution is a 182-dimensional representation that combines pooled node attributes, two-step WL neighborhood signatures, and explicit global graph descriptors. On the retained 235-graph dataset, XGBoost reaches 78.81% balanced accuracy and 0.9033 ROC-AUC, while Random Forest remains statistically competitive. The feature-component analysis indicates that pooled node information provides a strong predictive baseline and that adding WL neighborhoods is associated with improved observed performance.

The experiment also identifies an important limitation: performance falls under the node/edge-signature stress split. Because that grouping is not a verified crate- or advisory-level partition, it cannot be interpreted as cross-project validation. The result shows that the reported performance is sensitive to how repeated graph-size signatures are distributed within the dataset. Accordingly, LLVM-IR-derived CPG embeddings demonstrate useful discrimination on the evaluated data, while generalization to unseen Rust software requires separate evaluation.

The evidence reported here should be read as a within-dataset feasibility result for a compact CPG representation whose components can be evaluated separately, not as a claim that this representation is superior to source- or token-level alternatives; establishing that would require a matched comparison on shared datasets, folds, and metrics, which is outside the scope of this study.

Statements and Declarations

Funding

No funding was received for conducting this study.

Author contributions

Syeda Jannatul Boshra performed the methodology development, data collection and preparation, software implementation, experiments, formal analysis, validation, visualization, and preparation of the original manuscript. Young Lee contributed to the conceptualization of the study, supervision, methodological guidance, and substantial review and editing of the manuscript. Both authors read and approved the final manuscript.

Competing interests

The authors declare that they have no conflict of interest.

Ethics approval

The study uses publicly disclosed software materials and security advisories. It involves no human participants, personal data, or animal subjects.

Consent to participate

Not applicable.

Consent for publication

Not applicable.

Data availability

The data and processed artifacts used in this study are available through the anonymized replication package at <https://anonymous.4open.science/r/anonymou-s-rust-2165/>.

Code availability

The processing and analysis code supporting the experimental workflow, including LLVM IR generation, CPG processing, feature extraction, model evaluation, and result generation, is available through the anonymized replication package at <https://anonymous.4open.science/r/anonymou-s-rust-2165/>.

References

- [1] Matsakis, N.D., Klock, F.S.: The rust language. *ACM SIGAda Ada Letters* **34**(3), 103–104 (2014) <https://doi.org/10.1145/2692956.2663188>
- [2] Jung, R., Jourdan, J.-H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages* **2**(POPL), 1–34 (2018) <https://doi.org/10.1145/3158154>
- [3] Rust Secure Code Working Group: RustSec Advisory Database. <https://github.com/RustSec/advisory-db>. Accessed 15 July 2026 (2026)
- [4] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: *International Symposium on Code Generation and Optimization*, pp. 75–86. IEEE, Piscataway, NJ, USA (2004). <https://doi.org/10.1109/CGO.2004.1281665>
- [5] Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: *2014 IEEE Symposium on Security and Privacy*, pp. 590–604. IEEE Computer Society, Los Alamitos, CA, USA (2014). <https://doi.org/10.1109/SP.2014.44>
- [6] Fraunhofer AISEC: Code Property Graph Framework. <https://github.com/Fraunhofer-AISEC/cpg>. Accessed 15 July 2026 (2026)
- [7] Cui, M., Chen, C., Xu, H., Zhou, Y.: Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *arXiv preprint arXiv:2103.15420* (2021)
- [8] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. In: *Proceedings of the Network and Distributed System Security Symposium*. Internet Society, Reston, VA, USA (2018). <https://doi.org/10.14722/ndss.2018.23158>
- [9] Fu, M., Tantithamthavorn, C.: Linevul: A transformer-based line-level vulnerability prediction. In: *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 608–620. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3524842.3528452>
- [10] Lee, Y., Boshra, S.J., Yang, J., Cao, Z., Liang, G.: Machine learning-based vulnerability detection in rust code using llvm ir and transformer model. *Machine Learning and Knowledge Extraction* **7**(3) (2025) <https://doi.org/10.3390/make7030079>
- [11] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: *Advances in Neural Information Processing Systems*, vol. 32 (2019). <https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html>
- [12] Li, Y., Wang, S., Nguyen, T.N.: Vulnerability detection with fine-grained interpretations. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 292–303. Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3468264.3468597>
- [13] Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* **48**(9), 3280–3296 (2022) <https://doi.org/10.1109/TSE.2021.3087402>
- [14] Shervashidze, N., Schweitzer, P., Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler–lehman graph kernels. *Journal of Machine Learning Research* **12**, 2539–2561 (2011)
- [15] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* **19**(4), 2244–2258 (2022) <https://doi.org/10.1109/TDSC.2021.3051525>

- [16] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, É.: Scikit-learn: Machine learning in python. *Journal of Machine Learning Research* **12**, 2825–2830 (2011)
- [17] Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794. Association for Computing Machinery, New York, NY, USA (2016). <https://doi.org/10.1145/2939672.2939785>
- [18] Breiman, L.: Random forests. *Machine Learning* **45**(1), 5–32 (2001) <https://doi.org/10.1023/A:1010933404324>
- [19] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: *International Conference on Learning Representations* (2017). <https://openreview.net/forum?id=SJU4ayYgl>
- [20] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: *Advances in Neural Information Processing Systems*, vol. 32 (2019)
- [21] Fey, M., Lenssen, J.E.: Fast graph representation learning with pytorch geometric. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). <https://arxiv.org/abs/1903.02428>
- [22] Chicco, D., Jurman, G.: The advantages of the matthews correlation coefficient over f1 score and accuracy in binary classification evaluation. *BMC Genomics* **21** (2020) <https://doi.org/10.1186/s12864-019-6413-7>
- [23] Saito, T., Rehmsmeier, M.: The precision–recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE* **10**(3), 0118432 (2015) <https://doi.org/10.1371/journal.pone.0118432>