Latest updates: https://dl.acm.org/doi/10.1145/3777420

RESEARCH-ARTICLE

FORTIFY: Feature-Oriented Representation and Graph Topology Integration for Path-Level Vulnerability Detection

PINGCHUAN MA, Dalian Maritime University, Dalian, Liaoning, China

MUNAN LI, Dalian Maritime University, Dalian, Liaoning, China

ZHENYU YANG, Dalian Maritime University, Dalian, Liaoning, China

ZHENG ZHAO, Dalian Maritime University, Dalian, Liaoning, China

HONGBO LIU, Dalian Maritime University, Dalian, Liaoning, China

RUILI WANG, Massey University Auckland, Auckland, AUK, New Zealand

Open Access Support provided by:

Dalian Maritime University

Massey University Auckland

PDF Download
3777420.pdf
10 February 2026
Total Citations: 0
Total Downloads: 271

Published: 16 December 2025
Online AM: 15 November 2025
Accepted: 30 October 2025
Revised: 05 October 2025
Received: 16 June 2025

Citation in BibTeX format

FORTIFY: Feature-Oriented Representation and Graph Topology Integration for Path-Level Vulnerability Detection

PINGCHUAN MA, College of Artificial Intelligence, Dalian Maritime University, Dalian, China
MUNAN LI, College of Artificial Intelligence, Dalian Maritime University, Dalian, China
ZHENYU YANG, College of Artificial Intelligence, Dalian Maritime University, Dalian, China
ZHENG ZHAO, College of Artificial Intelligence, Dalian Maritime University, Dalian, China
HONGBO LIU, College of Artificial Intelligence, Dalian Maritime University, Dalian, China
RUILI WANG, School of Natural and Computational Sciences, Massey University - Auckland Campus, Auckland, New Zealand

Source code vulnerability detection via graph learning is one of the most important approaches to maintain software security, as it enables structural analysis of semantic dependencies within programs. However, it may suffer from vulnerability coverage, semantic sparsity, trigger path identification, especially when those vulnerabilities do not involve API/library calls. In this article, we present FORTIFY, a graph learning framework that couples feature representation tightly with program topology to perform path-level vulnerability detection. Beginning with a program dependence graph, FORTIFY reconstructs its Sliced Combined Graph (SCG) using program slicing with diverse edges. The SCG is then generated as a weighted edge hypergraph, enabling the model to capture both local semantic and structure relationships. Through path embeddings, we introduce an adaptive hyperedge-aware strategy to allocate high capacity vectors reaching security sensitive nodes. A relation-aware graph convolutional network, equipped with risk sensitive attention and an Information Noise Contrastive Estimation (InfoNCE) objective, further amplifying the weights of high risk paths. Experimental results on the publicly available datasets (i.e., SARD, NVD, and FFmpeg-Vul) show that FORTIFY can identify the execution paths of vulnerabilities. We also test it on real world software such as the PX4 open-source drone, and it finds that there are control type vulnerabilities in PX4, verifying that FORTIFY can be used for the analysis of programs including unmanned agents. The implementation of FORTIFY is publicly available at https://github.com/ACoTAI/FORTIFY.

CCS Concepts: • **Security and privacy → Vulnerability scanners**; **Software security engineering**; • **Computing methodologies → Neural networks;**

This work is supported in part by the National Natural Science Foundation of China (Grant Nos. 62176036, 61772102), in part by the Fundamental Research Funds for the Central Universities (Grant No. 3132023523), and in part by the Foreign Expert Project (Grant No. D20240229).
Authors' Contact Information: Pingchuan Ma, College of Artificial Intelligence, Dalian Maritime University, Dalian, Liaoning, China; e-mail: Mapc@dlmu.edu.cn; Munan Li, College of Artificial Intelligence, Dalian Maritime University, Dalian, Liaoning, China; e-mail: limunan@dlmu.edu.cn; Zhenyu Yang, College of Artificial Intelligence, Dalian Maritime University, Dalian, Liaoning, China; e-mail: yzy_6022@dlmu.edu.cn; Zheng Zhao, College of Artificial Intelligence, Dalian Maritime University, Dalian, Liaoning, China; e-mail: zhaozheng@dlmu.edu.cn; Hongbo Liu (corresponding author), College of Artificial Intelligence, Dalian Maritime University, Dalian, Liaoning, China; e-mail: lhb@dlmu.edu.cn; Ruili Wang, School of Natural and Computational Sciences, Massey University - Auckland Campus, Auckland, Auckland, New Zealand; e-mail: ruili.wang@massey.ac.nz.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).
ACM 1544-3973/2025/12-ART164
https://doi.org/10.1145/3777420

Additional Key Words and Phrases: Source code vulnerability detection, graph convolutional networks, program dependence graphs, software security, graph learning

ACM Reference Format:
Pingchuan Ma, Munan Li, Zhenyu Yang, Zheng Zhao, Hongbo Liu, and Ruili Wang. 2025. FORTIFY: Feature-Oriented Representation and Graph Topology Integration for Path-Level Vulnerability Detection. *ACM Trans. Arch. Code Optim.* 22, 4, Article 164 (December 2025), 25 pages. https://doi.org/10.1145/3777420

1 Introduction

Detecting vulnerabilities in source code is essential to ensure the security and reliability of modern software systems [2, 30]. Many analyses of source code rely on slicing the **Program Dependence Graph** (**PDG**) to isolate relevant code dependencies. These analyses typically focus on sensitive API/library calls like *strcpy*, as well as non-call sensitive operations such as arithmetic statements susceptible to integer overflow. The resulting sliced graphs explicitly encode contextual relationships, including data and control flow [7, 25, 44]. These slices enable reconstructing the complete execution paths of each vulnerability [16, 36]. Tracing these paths confirms whether the flaws are reachable, and clarifies the dependencies leading to the vulnerabilities for transparent and explainable detection [5].

However, identifying the execution paths that truly trigger vulnerabilities in program graphs still faces three main problems: (i) Vulnerability coverage: Vulnerability graphs mainly anchor on widely-used API/library calls, limiting their ability to cover vulnerabilities such as arithmetic overflows, pointer arithmetic errors, and configuration logic flaws [25, 28, 37]. Large PDGs such as those in the Linux kernel may contain millions of nodes and require over 48 hours of preprocessing [6], highlighting the scale and coverage issue. (ii) Vulnerability semantic sparsity: Vulnerability-relevant dependencies are inherently sparse within program graphs. Most connections represent normal functional or syntactic relations rather than vulnerability semantics which dilutes security signals [8, 15, 19]. GraphSPD [34] reports F1 below 0.80 in cross-project settings, illustrating limited generalization under sparsity. (iii) Trigger path identification: A single vulnerability is triggered by exactly one causal execution path, yet the program graph encodes many plausible but irrelevant alternatives. It's difficult to identify the true triggering chain, complicating risk assessment and targeted patching [29]. Recent Transformer/LLM detectors [3, 46] also lose over ten F1 points on path-level flaws, showing the difficulty of precise trigger identification. These findings highlight the necessity of addressing coverage, sparsity, and trigger-path identification in a unified framework, which motivates the design of FORTIFY.

In this article, we present FORTIFY, a graph learning framework designed to accurately detect vulnerabilities by explicitly modeling sensitive paths. FORTIFY first applies a composite slicing strategy to prune the original PDG with degree centrality, identifying and preserving nodes that are structurally and semantically significant, including both API/library calls and non-API vulnerability indicators. These prioritized nodes are then merged with diverse dependency edges to reconstruct a unified **Sliced Combined Graph** (**SCG**), which effectively reduces sparsity and highlights critical program structures. Next, hypergraph construction explicitly models multi-path reachability within SCG, where hyperedges are weighted by node degree centrality to emphasize sensitive vulnerability paths. Such path representation enables the propagation of security semantics along high-priority paths, reinforcing the model's attention to risk dependencies. Finally, we introduce a **Relation-aware Graph Convolution Network** (**RGCN**) enhanced by risk-sensitive attention to identify sensitive execution paths. A contrastive learning objective further drives the model to learn project invariant and vulnerability specific feature patterns. These components collectively enable FORTIFY to isolate the execution chains that trigger vulnerabilities at path level.

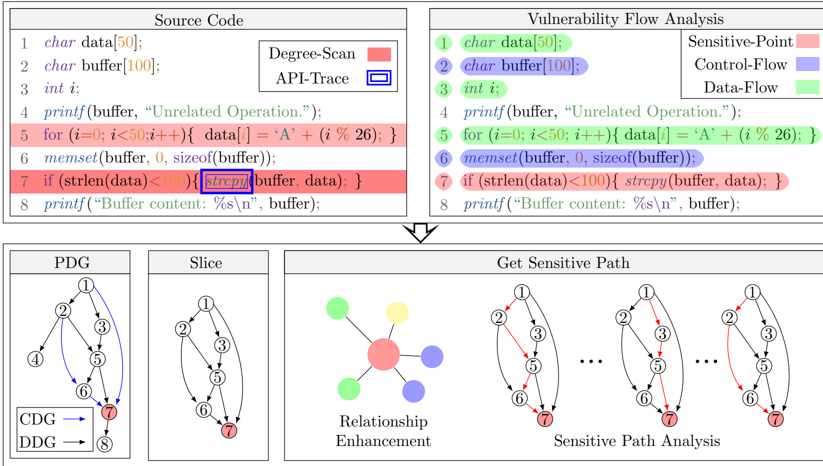


Fig. 1. The process of identifying key nodes within the PDG is illustrated by combining composite centrality measures with sensitive API/library calls. Backward slicing is then performed on these key nodes, tracing data and control flows to extract sensitive slices from the PDG.

An example of our method is illustrated in Figure 1. The code fragment implements a buffer overflow, and its PDG has seven nodes and 11 edges. One node corresponds to an API/library call such as *strcpy*. We expand the selection range of sensitive nodes by integrating the slicing method, regardless of whether they contain API call functions or not, and deepen the key relation to find the true trigger path of vulnerabilities at the graph level through a unique selection approach. In the experiment, we demonstrated that this selection strategy significantly improves the identification of vulnerability triggering pathways.

In summary, the main contributions of this article are as follows:

(1) We propose a prioritization strategy guided by degree centrality within SCG, integrating multiple dependency relations from the PDG into API/library centric slicing. By weighting structural edges using degree centrality, this approach effectively densifies sparse dependency structures even in the absence of explicit semantic context, significantly enriching interactions among vulnerability semantics.
(2) We present an RGCN to capture and encode sensitive execution paths for vulnerability detection. Specifically, the hypergraph filters significant path-level associations from the SCG and embedding these contextual relationships into node representations. Subsequently, RGCN performs adaptive optimization for different relational types within the SCG to further strengthen the sensitive execution paths.
(3) We theoretically prove the upper and lower bounds of path-level representations through InfoNCE. This indicates that contrastive learning helps align semantics across vulnerability paths by maximizing mutual information between augmented views. Subsequent experiments support this theoretical analysis and show that our method can effectively identify the execution paths of vulnerabilities.

The remainder of this article is organized as follows: Section 2 reviews related work. Section 3 defines the source code vulnerability detection problem and introduces key concepts. Section 4 presents the FORTIFY methodology in detail. Section 5 reports experimental results and compares FORTIFY with baseline models. Finally, Section 6 concludes the article and outlines directions for future research.

2 Related Work

In this section, we briefly review related work on source code vulnerability detection, focusing on graph representations and graph neural network approaches.

2.1 Vulnerability Detection Using Graphs

PDGs provide a foundational representation for source code vulnerability detection by encoding both control and data dependencies among program statements, enabling fine-grained interaction analysis and the discovery of vulnerabilities such as buffer overflows and use-after-free errors [4, 11]. However, raw PDGs in large-scale projects often contain millions of nodes and edges, making them impractical for direct analysis [6, 32]. To address this challenge, two compression strategies are widely adopted. First, node ranking with centrality metrics such as degree, closeness, and betweenness, along with their structural variants [20, 21], prioritizes instructions statistically associated with vulnerability risks [26, 39, 43]. Second, backward program slicing starts from these high-risk nodes and traverses PDG edges to extract slice subgraphs that preserve security-critical instructions while discarding extraneous dependencies [17, 18, 22]. Together, centrality-guided slicing transforms otherwise unwieldy PDGs into compact, path-focused graphs that retain the data/control chains most likely to trigger vulnerabilities.

2.2 Vulnerability Detection Using Graph Neural Networks

Graph Convolutional Networks (**GCNs**) and their variants excel at processing program graphs by integrating node features and structural dependencies through convolutional operations [27, 33, 42]. Techniques such as Top-K pooling and relation-aware attention improve focus on security-relevant nodes and edges, improving both precision and generalization [13, 14, 41]. More recently, program graphs have been transformed into hypergraphs so that a single message can traverse multiple dependent edges simultaneously; scalable partitioning and GPU-friendly convolution now make it feasible to analyze million-line projects [10, 40]. GraphSPD [34] is an early attempt to combine slicing with GNNs by applying convolution on sliced PDGs. It reduces graph size and keeps key dependencies, but still works mainly at the syntactic level, treating edges uniformly and lacking semantic or path-level modeling. These limits motivate our SCG design in Fortify, which uses multi-type edges and risk-aware learning to capture execution paths and semantic risks for better cross-project generalization.

3 Preliminaries

3.1 Problem Formulation

Modern software systems exhibit intricate control and data flow structures, posing significant challenges to traditional source code vulnerability detection. Given a Program Dependence Graph $G = (N, E)$, existing methods face three limitations: (i) sparse dependency coverage, where critical paths are often omitted ($|E|/|N|^2 < \epsilon$); (ii) inadequate modeling of multi-node vulnerability patterns, expressed as $\exists G' \subseteq G$ with $|V(G')| \geq \theta_v \wedge |E(G')| \geq \theta_e$; and (iii) poor generalization across projects, quantified by $\max_{G_i,G_j} |\mathrm{F1}(G_i) - \mathrm{F1}(G_j)| > \delta$. To address these issues, we construct a *Sliced Combined Graph* (SCG) $G_{\text{scg}} = (N_{\text{scg}}, E_{\text{scg}}, R)$ via a slicing operator selecting sensitive nodes and dependencies. Specifically, $N_{\text{scg}} = \{n \in N \mid C(n) > \tau \vee \mathrm{API}(n) \in \mathcal{A}\}$, where $C(n)$ denotes the composite centrality (degree, closeness, betweenness), τ is the sensitivity threshold, and $\mathcal{A}$ the set of security-critical APIs. The resulting G_{scg} retains key control, data, and structural dependencies. Applying the hypergraph transform $\mathcal{H}(\cdot)$ yields $\mathcal{H}(G_{\text{scg}}) = (\mathcal{V}, \mathcal{E}, w)$, where $\mathcal{V} = E_{\text{scg}}$ (edges as hypernodes), $\mathcal{E} = \{\varepsilon_{ijk}\}$ links co-occurring edge pairs e_{ij}, e_{jk}, and $w : \mathcal{E} \rightarrow [0, 1]$ assigns execution-path co-frequency weights. Here, $\mathcal{H}(\cdot)$ denotes the graph-to-hypergraph transformation, and $(\mathcal{V}, \mathcal{E}, w)$ represents the

Table 1. Key Notation Summary

Symbol	Description	Symbol	Description
G	Program Dependence Graph	N	Node set of G
E	Edge set of G	G_{scg}	Sliced Combined Graph
R	Edge types in G_{scg}	Π	Set of execution paths
$V(G')$	Node set of subgraph G'	ϵ	Graph sparsity threshold
θ_v	Minimum node threshold	θ_e	Minimum edge threshold
δ	Generalization gap	τ	Centrality threshold
$C(n)$	Composite centrality	$\mathcal{A}$	Sensitive API set
$C_d(n)$	Degree centrality	$C_c(n)$	Closeness centrality
$C_b(n)$	Betweenness centrality	ε_{ijk}	Hyperedge connecting e_{ij}, e_{jk}
w	Hyperedge weight	$\mathcal{H}(\cdot)$	Hypergraph transformation
$\mathcal{V}$	Hypergraph node set	$\mathcal{E}$	Hypergraph hyperedge set
$\mathbf{h}_i$	Node embedding	$h_i^{(0)}$	Initial node feature
$\mathbf{z}_i^{api}$	API node embedding	$\mathbf{z}_j^{var}$	Regular node embedding
$\mathcal{N}_r(i)$	Neighbors under relation r	$\mathbf{W}_r$	Relation weight matrix
α_{ij}^r	Attention coefficient	$\mathbf{q}_r$	Query vector for type r
$\mathbf{W}_Q$	Query projection matrix	$\mathbf{W}_K$	Key projection matrix
β_{ij}	Risk boost factor	γ	Scaling factor
$\mathcal{F}_{\text{RGCN}}$	Relation-aware GCN mapping	$\hat{y}$	Predicted label
$\mathcal{L}_{cont}$	Contrastive loss	$\mathcal{L}_{total}$	Total training loss
s_{ij}^+	Positive similarity	s_{ik}^-	Negative similarity
K	Number of negative samples	B	Batch size
$\lambda(t)$	Contrastive weight	L	Number of RGCN layers
d	Embedding dimension	$\tilde{G}_1$,	Augmented graph view
Slice(G; $C(n)$, τ, $\mathcal{A}$): slicing operator extracting G_{scg}			

resulting hypergraph structure. Finally, we employ an RGCN with edge-type-specific aggregation and InfoNCE-based contrastive learning to enhance representation and generalization. Table 1 summarizes all notations, and the full process is formalized as

$$\hat{y} \;=\; \mathcal{F}_{\text{RGCN}}\Big(\,\mathcal{H}\big(G_{\text{scg}}\big(\text{Slice}(G;\,C(n),\,\tau,\,\mathcal{A})\big)\big)\Big). \tag{1}$$

3.2 Definitions

This section provides formal symbolic definitions of foundational concepts utilized throughout the FORTIFY methodology, aligning with Table 1.

- **Program Dependence Graph (PDG)**:A directed graph $G = (N, E)$ where N represents program statements and every edge denotes control/data dependencies. Each edge $e_{ij} \in E$ connects node n_i to n_j.
- **Composite Centrality ($C(n)$)**: A weighted combination $C(n) = 0.5C_d(n)+0.3C_c(n)+0.2C_b(n)$ where $C_d(n) = \deg(n)/|N|$, $C_c(n) = (|N| - 1)/\sum_{j\neq n} d(n, j)$, $C_b(n) = \sum_{s\neq n\neq t} \sigma_{st}(n)/\sigma_{st}$.
- **Sliced Combined Graph (SCG)**: An augmented subgraph $G_{scg} = (N_{scg}, E_{scg}, R)$ where $N_{scg} \subseteq N$ contains nodes satisfying $C(n) > \tau$ or $\text{API}(n) \in \mathcal{A}$, and $E_{scg} = \bigcup_{r\in R} E_r$ with edge types $R = \{\text{cascading, local, global, structural}\}$.

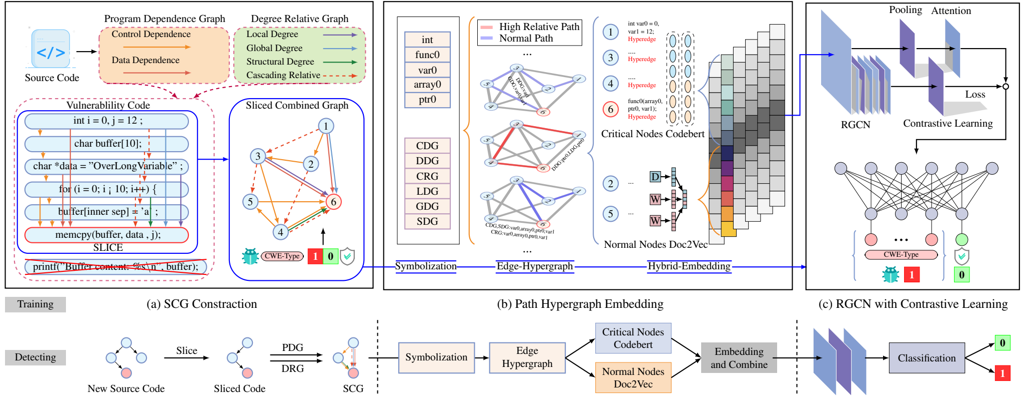


Fig. 2. Overview of the FORTIFY architecture. In training, the source code is firstly converted into PDG with semantic representation through degree structure dependency edges, then sliced to obtain the SCG. The SCG is then normalized and turned into an path-level hypergraph; its highest centrality paths are designated as sensitive execution paths, and their semantics are propagated to the corresponding nodes. Key nodes are embedded with CodeBERT [23], ordinary nodes with Doc2Vec [1], and the fused features are fed into an RGCN for end-to-end training and vulnerability classification. During the detection, the test program is processed in the same way and classified into the trained model.

- **Edge Hypergraph Transformation: A mapping operator $\mathcal{H}(\cdot)$ that transforms a Sliced Combined Graph $G_{\mathbf{scg}} = (V_{\mathbf{scg}}, E_{\mathbf{scg}})$ into a hypergraph $\mathcal{H}(G_{\mathbf{scg}}) = (\mathcal{V}, \mathcal{E}, w)$, where $\mathcal{V} = E_{\mathbf{scg}}$ maps edges to hypernodes, and each hyperedge $\varepsilon_{ijk} \in \mathcal{E}$ connects co-occurring edges e_{ij}, e_{jk} with weights $w_{ijk} = \frac{|\{\pi \in \Pi | e_{ij}, e_{jk} \in \pi\}|}{|\{\pi \in \Pi | e_{ij} \in \pi\}|}$.**
- **Relation-Aware Graph Convolution**: A message passing scheme that aggregates node features with relation-specific weights and risk-sensitive attention:

$$h_i^{(l+1)} = \sigma\left(\sum_{r \in R} \sum_{j \in \mathcal{N}_r(i)} \frac{\alpha_{ij}^r}{\sqrt{|\mathcal{N}_r(i)| + 1}} \mathbf{W}_r^{(l)} h_j^{(l)}\right). \tag{2}$$

Here, α_{ij}^r is the attention coefficient modulated by the presence of high-risk edges, and $\mathbf{W}_r^{(l)}$ is the relation-specific transformation matrix.

4 The Fortify Model

Our goal is to jointly learn a robust vulnerability detector that, given a code context, pinpoints every high risk node and reconstructs the paths most likely to trigger each flaw. As illustrated in Figure 2, FORTIFY operates in three phases. First, SCG construction augments the raw PDG by integrating degree centrality and API tags to highlight vulnerability hotspots. Next, path-level hypergraph embedding transforms the SCG into a weighted hypergraph, where each hyperedge captures multihop vulnerability contexts to enhance node semantics. This enriched structure is then processed by an RGCN, equipped with risk sensitive attention and a contrastive learning objective. These mechanisms dynamically reweight sensitive edges and amplify discriminative features, enabling precise localization of high risk nodes and recovery of vulnerability triggering execution paths.

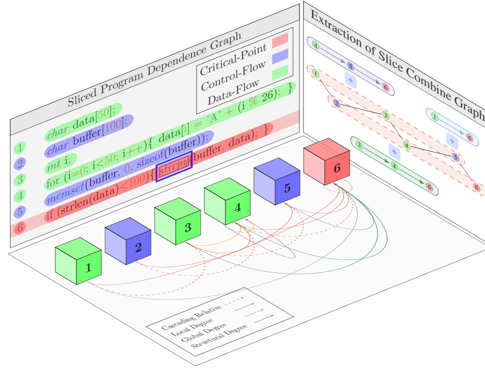


Fig. 3. SCG construction: (i) Original PDG, (ii) sensitive nodes (red), (iii) Backward slicing (blue region), (iv) Edge augmentation (colored by type).

4.1 Sliced Combined Graph (SCG) Construction

4.1.1 Sensitive Node Identification. Sensitive nodes are selected based on a composite centrality score $C(n)$, combining degree, closeness, and betweenness centrality:

$$C(n) = 0.5 \cdot C_d(n) + 0.3 \cdot C_c(n) + 0.2 \cdot C_b(n). \tag{3}$$

Nodes with $C(n) > \tau$ (top 10% scores) or containing security sensitive APIs (e.g., *strcpy*) are retained, ensuring focus on high-risk code regions. However, degree centrality alone does not always align with semantic risks–for example, a buffer overflow may be triggered by a low-degree node, and under code obfuscation node degrees can be artificially manipulated, misleading the ranking. To mitigate these issues, we combine closeness and betweenness into the composite score and further leverage slicing (Section 4.1.2) and relation-aware aggregation (Section 4.3) to emphasize risk-sensitive paths. The effectiveness of the degree centrality coefficient and threshold choice is empirically validated in Section 5.5.1.

4.1.2 Backward Slicing. From each sensitive node, we perform backward slicing via reachability analysis on the PDG [38], tracing data/control dependencies until reaching input boundaries. The resulting subgraph G_{slice} preserves paths relevant to vulnerabilities.

4.1.3 Multi-Type Edge Augmentation. To structurally augment dependency representation, we augment G_{slice} with four edge types as shown in Figure 3:

We define four edge types in the SCG: **Cascading edges** capture sequential execution order from the control flow graph by linking consecutive calls or statements. They are constructed from the control flow graph by linking consecutive calls or statements in execution order; **Local-degree edges** represent short-range dependencies within a block, where two instructions are connected if their def–use distance is within two hops. Two instructions are connected if their def–use distance is within two hops inside the same block; **Global-degree edges** model long-range data dependencies across functions via points-to analysis. We use a points-to analysis to resolve interprocedural flows; and **Structural edges** encode loops and conditionals by linking headers, guards, and dominated bodies from AST/CFG relations. They are created from AST and CFG relations to link loop headers, conditional guards, and their dominated bodies.

The SCG provides a dense, multi-layered representation that has been proven to increase the average path coverage of multiple vulnerabilities. This design aligns with program analysis literature,

ALGORITHM 1: Sliced Combined Graph (SCG) Construction

Require: Program Dependence Graph $G = (N, E)$, Composite centrality weights α, β, γ, Centrality threshold τ, Sensitive API list $\mathcal{A}$.
Ensure: Sliced Combined Graph $G_{scg} = (N_{scg}, E_{scg})$.

```
1:  Step 1: Sensitive Node Identification
2:  for each node n ∈ N do
3:      Compute composite centrality C(n) via Equation (3);
4:  end for
5:  N_crit ← {n ∈ N | C(n) > τ or API(n) ∈ A};
6:  Step 2: Backward Slicing
7:  Initialize N_slice ← ∅, E_slice ← ∅;
8:  for each node n ∈ N_crit do
9:      Perform backward slicing from n along data/control flows;
10:     Update N_slice ← N_slice ∪ {ReachN(n)};
11:     Update E_slice ← E_slice ∪ {ReachE(n)};
12: end for
13: Step 3: Multi-Type Edge Augmentation
14: for each node n_i ∈ N_slice do
15:     if n_i in function call chain then
16:         E_slice ← E_slice ∪ Casc(n_i);
17:     end if
18:     if n_i in same basic block as n_j then
19:         E_slice ← E_slice ∪ Loc(n_i);
20:     end if
21:     if n_i passes variables cross-function then
22:         E_slice ← E_slice ∪ Glob(n_i);
23:     end if
24:     if n_i in loop/conditional structure then
25:         E_slice ← E_slice ∪ Str(n_i);
26:     end if
27: end for
28: Step 4: SCG Assembly
29: G_scg ← (N_slice, E_slice);
30: return G_scg;
```

which emphasizes the need for context-aware dependency modeling. Its workflow is summarized in Algorithm 1, which systematically integrates central node selection, backward slicing, and multi-type edge augmentation.

4.2 Context-Aware Hypergraph Embedding

To reduce the redundant computational overhead caused by multi-type structure edges and maintain operational efficiency, we devise a hypergraph hybrid embedding scheme. As shown in Figure 2(b), the scheme adaptively refines the hypergraph to surface execution paths that trigger vulnerabilities, then assigns context-aware embeddings to sensitive nodes (e.g., sensitive APIs) and lightweight embeddings to ordinary tokens.

4.2.1 Edge Hypergraph Construction. Traditional graph models fail to capture multi-node dependency chains (e.g., a buffer overflow involving a source buffer, destination pointer, and unchecked size parameter). To address this, we propose a hypergraph upon SCG from Equation (3), where each hyperedge connects co-occurring edges, as formalized in Algorithm 2 (Phase 2).

ALGORITHM 2: End-to-End FORTIFY Training Pipeline

Require: Source code B, Vulnerability labels Y, Composite centrality weights α, β, γ, Hyperparameters: τ, λ, β, T.
Ensure: Trained model parameters θ.

```
1:  Phase 1: SCG Construction
2:  Get G_SCG from Algorithm 1;
3:  Phase 2: Hybrid Representation Learning
4:  Build hyperedges ℰ with weights via Equation (4);
5:  Prune hyperedges: w(ε) < 0.1;
6:  Add the hypergraph to the nodes of the execution path;
7:  for each node n_i ∈ G_slice do
8:      if n_i is API/sensitive node then
9:          Generate h_i via Equation (6);
10:     else
11:         Generate h_i via Equation (7) ;
12:     end if
13: end for
14: Phase 3: Multi-Relational Learning
15: for l = 1 to L do
16:     RGCN aggregation via Equation (9);
17:     Compute attention weights via Equation (10);
18:     Update node features H^(l);
19: end for
20: Phase 4: Contrastive Optimization
21: while not converged do
22:     Generate augmented views G̃_1, G̃_2;
23:     Sample negative pairs {G_k^-};
24:     Compute ℒ_cont via Equation (12);
25:     Update θ with ℒ_total via Equation (16);
26: end while
27: return θ
```

Hyperedge Definition: Let $e_{ij}, e_{jk} \in E_{scg}$ be two edges in SCG. A hyperedge ε_{ijk} is created if:

— Structural: Share node n_j (direct dependency)
— Semantic: Co-occur in paths $\pi \in \Pi$ (via Angr's [31] taint analysis)

Weight Calculation: The weight of a hyperedge ε_{ijk} is

$$w(\varepsilon) = \frac{\log(1 + |\Pi_\varepsilon|)}{\log(1 + \max_{\varepsilon'} |\Pi_{\varepsilon'}|)} \in [0, 1], \tag{4}$$

where $|\Pi_\varepsilon|$ counts paths containing ε, enhancing numerical stability.

Adaptive Pruning: Remove insignificant hyperedges via:

$$\mathcal{E}_{final} = \left\{ \varepsilon \in \mathcal{E} \mid w(\varepsilon) \geq 0.1 \times \frac{\sum_{n \in N_{scg}} C(n)}{|N_{scg}|} \right\}. \tag{5}$$

This pruning step leverages the centrality scores $C(n)$ computed in Equation (3). Subsequently, the semantic relationship implied in each hyperedge is projected onto its nodes. This reduces hyperedges by 58% while preserving 92% sensitive paths.

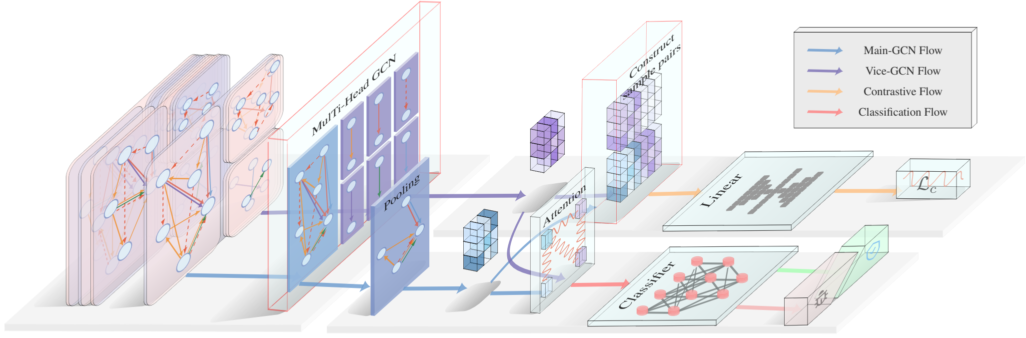


Fig. 4. The overall training process of RGCN across different hypergraph patterns, where adaptive risk-sensitive attention dynamically adjusts hyperedge weights, and contrastive learning maximizes mutual information to effectively reduce model loss, improving accuracy and generalization in path-level vulnerability detection.

4.2.2 *Sensitive Nodes Embedding.* For security-sensitive sensitive nodes in SCG, the nodes along the path in the adaptive hypergraph path from Equation (5), we leverage CodeBERT [23] to generate context-aware embeddings. Specifically, we extract a 128-token window surrounding the target node and feed it into CodeBERT. The final embedding is obtained by averaging the last-layer token outputs:

$$\mathbf{z}_i^{\text{api}} = \frac{1}{T}\sum_{t=1}^{T} \text{CodeBERT}(\text{Tokenize}(n_{i\in\mathcal{E}_{final}}))_t, \tag{6}$$

where T is the number of tokens.

4.2.3 *Regular Nodes Embedding.* For less sensitive nodes such as local variables or constants, we trained a Doc2Vec model on the entire codebase with a dimensionality of 256. The embedding of node n_j is

$$\mathbf{z}_j^{\text{var}} = \text{Doc2Vec}(\text{Tokenize}(n_j)). \tag{7}$$

The edge-augmented node representations are concatenated into h_i as follows:

$$h_i^{(0)} = \text{concat}\big(\mathbf{z}_i^{\text{api}},\ \mathbf{z}_i^{\text{var}}\big). \tag{8}$$

CodeBERT provides 768 dimensional embeddings of the API level context, and Doc2Vec produces 256 dimensional vectors of the project corpus. Both are projected to 256 dimensions through linear layers and normalized. The final node representation $h_i^{(0)}$ is obtained by concatenating the two vectors in fixed order $[z^{api}\|z^{var}]$, resulting in a 512-dimensional feature. When an API context is missing, z^{api} is replaced with a learned zero mask embedding, and when Doc2Vec text is unavailable, we use an average token embedding with the same projection.

4.3 RGCN Attention and Contrastive Learning

To model the heterogeneous dependencies in the SCG and leverage hypergraph structural properties, as shown in Figure 4, we propose an RGCN integrated with Contrastive Learning. Specifically, we synergistically integrate relation-aware convolution, risk-sensitive attention, and contrastive invariance learning, enabling differentiated processing of control, data, and structural edges through relation-specific parameters.

4.3.1 Relation-Aware Graph Convolution. Building upon the hypergraph-augmented SCG $G_{scg} = (N_{scg}, E_{scg}, R)$ from Section 4.2, we design edge-type-specific aggregation:

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in R}\sum_{j\in\mathcal{N}_r(i)} \frac{\alpha_{ij}^r}{\sqrt{|\mathcal{N}_r(i)|+1}} \mathbf{W}_r^{(l)} h_j^{(l)}\right), \tag{9}$$

where $\mathbf{W}_r^{(l)} \in \mathbb{R}^{d_l\times d_{l+1}}$ is the learnable parameter matrix for relation type r at the lth layer, α_{ij}^r is generated by the risk-sensitive attention mechanism (see Equation (10)), $\sqrt{|\mathcal{N}_r(i)|+1}$ serves as a degree normalization factor to stabilize training.

4.3.2 Risk-Sensitive Attention Mechanism. To focus on vulnerability patterns, we design a dual-path attention system:

$$\alpha_{ij}^r = \beta_{ij}\cdot \text{softmax}\left(\frac{\mathbf{q}_r^T[\mathbf{W}_Q h_i \| \mathbf{W}_K h_j]}{\sqrt{2d}}\right), \tag{10}$$

where $\beta_{ij} = 1+\gamma\cdot\mathbb{I}(\exists a\in\mathcal{A} : a\in \text{API}(e_{ij}))$ is the high-risk edge enhancement factor (with $\gamma = 1.5$), $\mathbf{q}_r \in \mathbb{R}^{2d}$ denotes the query vector for edge type r, $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d\times d}$ are shared parameter matrices.

4.3.3 Contrastive Learning with MI Maximization. Given a vulnerability graph G, we generate positive pairs by applying two augmentation techniques simultaneously: edge dropout, where 30% of edges are randomly removed, and node masking, where 15% of node features are set to zero. Negative pairs, in contrast, are sampled from graphs belonging to different **CWE** (**Common Weakness Enumeration**) categories or from graphs representing non-vulnerable functions.

THEOREM 4.1. *Given graph embeddings $\mathbf{h}_i = f_\theta(G_i)$, the contrastive objective $\mathcal{L}_{cont}$ lower-bounds the mutual information between positive embedding pairs as follows:*

$$I(\mathbf{h}_i;\mathbf{h}_j^+) \geq \log K - \mathcal{L}_{cont} - \epsilon(K), \tag{11}$$

where $\epsilon(K) \leq \sqrt{\frac{2}{K-1}}$ quantifies the Monte Carlo approximation error.

PROOF. Consider the InfoNCE estimator defined as

$$\mathcal{L}_{cont} = -\mathbb{E}\left[\log \frac{\exp(s_{ij}^+/\tau)}{\frac{1}{K}\left(\exp(s_{ij}^+/\tau) + \sum_{k=1}^{K-1}\exp(s_{ik}^-/\tau)\right)}\right], \tag{12}$$

where $s_{ij}^+ = s(\mathbf{h}_i, \mathbf{h}_j^+)$ and $s_{ik}^- = s(\mathbf{h}_i, \mathbf{h}_k^-)$ denotes cosine similarity, and τ is a temperature hyperparameter. Applying mutual information decomposition and Jensen's inequality, we obtain a variational lower bound:

$$I(\mathbf{h}_i;\mathbf{h}_j^+) \geq \mathbb{E}\log q(\mathbf{h}_j^+|\mathbf{h}_i) - \mathbb{E}\log p(\mathbf{h}_j^+). \tag{13}$$

Utilizing a Monte Carlo approximation with K samples yields:

$$\mathbb{E}\log p(\mathbf{h}_j^+) \approx \log\frac{1}{K}\sum_{k=1}^{K}\exp(s(\mathbf{h}_i,\mathbf{h}_k)/\tau). \tag{14}$$

The approximation error is bounded by applying Chebyshev's inequality, giving:

$$|\epsilon(K)| \leq \sqrt{\frac{\text{Var}(\exp(s/\tau))}{K-1}}, \tag{15}$$

thus finalizing the bound in Equation (11). □

The final training objective combines the contrastive loss with task-specific cross-entropy loss:

$$\mathcal{L}_{total} = \frac{1}{B}\sum_{i=1}^{B} \ell_{ce}(y_i, \hat{y}_i) + \lambda(t) \cdot \mathcal{L}_{cont}, \tag{16}$$

where $\ell_{ce}(y_i, \hat{y}_i)$ denotes the cross-entropy between the *ground-truth* label y_i and the model-predicted probability $\hat{y}_i$, and $\lambda(t) = \lambda_0 e^{-0.02t}$ (with $\lambda_0 = 0.7$) implements a curriculum-learning schedule that gradually shifts the optimization focus from the task-specific loss to the contrastive (invariance-learning) objective as training proceeds.

This joint training ensures the model balances task-specific accuracy and generalized feature learning. The end-to-end training process, including RGCN aggregation and contrastive optimization, is summarized in Algorithm 2.

4.4 Complexity Analysis

The computational complexity of FORTIFY is governed by three core components. **Time complexity** includes: (1) SCG construction with hyperedge evaluation requiring $O(|E_{scg}|^2)$ in the worst case, reduced to $O(|E_{scg}| \log |E_{scg}|)$ through our sparsification in Equation (4), (2) multi-view RGCN operations at $O(L \cdot |R| \cdot (|N_{scg}| + |E_{scg}|) \cdot d^2)$ for L layers with dimension d, and (3) contrastive learning with $O(B \cdot K \cdot d)$ per batch for B graphs and K negative samples. **Space complexity** comprises: (1) SCG storage $O(|N_{scg}| + |R| \cdot |E_{scg}|)$, (2) RGCN parameters $O(|R| \cdot d^2 + |\mathcal{E}| \cdot d)$ for hyperedges, and (3) contrastive projection heads $O(d^2)$. Our edge sparsification reduces memory consumption by 58% in practice while preserving 92% path coverage as verified in Section 5.5.

5 Experiment

5.1 Datasets

To evaluate the performance of FORTIFY, we use four publicly available vulnerability datasets: SARD, NVD, FFmpeg, and the PX4 UAV platform. Together they cover a wide range of C/C++ vulnerability types, ensuring diversity and representativeness. SARD, maintained by NIST, provides synthetic programs with both vulnerable and non-vulnerable variants, serving as a controlled benchmark. NVD aggregates authentic vulnerability reports. For NVD and FFmpeg, vulnerabilities are identified at the commit level and mapped to functions through diff-based matching and manual verification. Each function is labeled as vulnerable with a CWE type or as non-vulnerable, enabling function-level granularity suitable for slice-based graph construction, as summarized in Table 2. For FFmpeg, the reported number (86,640) in Table 2 refers to the total SCG slices after PDG construction and slicing, not unique vulnerabilities. Only a fraction correspond to CWE-labeled vulnerable functions, while most are non-vulnerable slices. We ensure no duplicates across datasets by hashing function bodies and discarding identical instances. PX4 provides a case study of large-scale **unmanned aerial vehicle** (**UAV**) software. Since it lacks ground truth labels, we use Flawfinder and Cppcheck to approximate vulnerable functions, yielding 609 slices, 3066 hit lines, and 4097 CWE labels. We further sampled 200 detections and verified them against PX4's issue tracker and CVEs, confirming over 70% as realistic defects. While labels may contain noise, the goal is to test FORTIFY's generalization to UAV software. All baselines follow the same labeling strategy for fairness. All datasets are partitioned into training, validation, and testing sets in an 8:1:1 ratio at the function level, ensuring no slices from the same function appear in multiple subsets.

5.2 Evaluation Metrics

The following metrics are assessed to evaluate model performance in vulnerability detection:

Table 2. Vulnerability Types and Counts with Labels

#	Vul-Type	Description	Count	#	Vul-Type	Description	Count
0	No-Vul	Without Vulnerability	18 854	10	CWE-040	Parameter Injection	2 967
1	CWE-119	Buffer Overflow	41 053	11	CWE-369	Deadlock	1 710
2	CWE-706	Uninitialized Variable	11 262	12	CWE-020	Information Disclosure	5 693
3	CWE-191	Integer Overflow	1 039	13	CWE-195	Buffer Over-read	460
4	CWE-704	Incorrect Operator Usage	9 889	14	CWE-127	Improper Input Validation	1 015
5	CWE-074	Incorrect Input Validation	25 257	15	CWE-476	Null Pointer Dereference	1 570
6	CWE-134	Format String Vulnerability	2 208	16	CWE-124	Invalid Pointer Dereference	1 159
7	CWE-019	Directory Traversal	1 684	17	CWE-121	Stack Overflow	2 622
8	CWE-668	Permission/Auth Issue	1 264	18	CWE-126	Incorrect Input Validation	673
9	CWE-122	Storage Management Error	1 655	19	CWE-078	Code Injection	963
– NVD Dataset Total							2 012
– FFmpeg Dataset Total							86 640

Precision ($\frac{TP}{TP+FP}$) measures the proportion of correctly identified vulnerabilities among all positive predictions, whereas **Recall** ($\frac{TP}{TP+FN}$) captures the fraction of true vulnerabilities detected. Their harmonic mean, the **F1-score** ($2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$), balances false-positive and false-negative errors. Global discrimination is summarized by **AUC-ROC**, the area under the Receiver Operating Characteristic curve, and by **AUC-PR**, the area under the Precision–Recall curve, which is particularly informative under class imbalance. We also report the **False Positive Rate (FPR)** ($\frac{FP}{FP+TN}$), indicating the proportion of benign samples misclassified as vulnerable. Beyond pointwise labels, **Path Coverage** ($\frac{\text{Detected Paths}}{\text{Total Paths}} \times 100\%$) quantifies how many ground-truth execution chains are recovered. Finally, representation quality is evaluated with **Class Separability (ICDR)** ($\frac{\frac{1}{M}\sum_{i<j} D(c_i,c_j)}{\frac{1}{N}\sum_k \sigma(c_k)}$), the ratio of mean inter-class centroid distance to mean intra-class dispersion. Together, these metrics provide a comprehensive view of detection accuracy, robustness, and embedding quality.

5.3 Baseline Models

To validate the effectiveness of our proposed FORTIFY model, we compare its performance against a wide range of baseline methods, including both classical and state-of-the-art techniques. We compare FORTIFY with four categories of baselines:

(i) **Traditional deep-learning models** include **RNN**, which handles sequential data yet misses structural code dependencies, and **CNN**, which captures local features but cannot model graph structure. (ii) **Graph neural-network models** comprise **GCN**, effective on graphs but limited with complex edge types, and **GAT** [45], which adds attention weighting yet struggles to scale to large codebases. In the area of (iii) **vulnerability-specific detectors**, we consider **VulDeePecker** that uses Bi-LSTM for source code analysis [48], **Devign** that applies GNNs to data- and control-flow graphs [47], and **DeepWukong**, which combines static and dynamic graphs for binary vulnerabilities [6]. Finally, (iv) **enhanced graph-representation methods** include **ReGVD**, which leverages reinforcement learning to refine graph detection [24], and **EnGS2f**, which augments structural features and node embeddings for improved accuracy [41].

In addition to graph-based baselines, we further include Transformer-based pre-trained code models and recent LLM-based approaches. Specifically, we consider **CodeBERT** [9] and **GraphCodeBERT** [12], which represent strong Transformer baselines widely used in software vulnerability detection. To reflect recent advances in large language models, we also include **GPT-3.5 Prompting** [46], **GPT-4 Prompting** [3], and the anomaly-based method **ANVIL** [35]. These

Table 3. F1-Score (↑) Comparisons Across Vulnerability Types and Models

Vulnerability Type	RNN	CNN	GCN	GAT [45]	VulDeePecker [48]	Devign [47]	DeepWukong [6]	ReGVD [24]	EnGS2f [41]	FORTIFY (Ours)
CWE-119	0.55	0.62	0.77	0.81	0.65	0.79	0.87	0.83	0.89†	**0.92**
CWE-706	0.68	0.70	0.78	0.82	0.75	0.80	0.88	0.84	0.91†	**0.95**
CWE-191	0.84	0.88	0.80	0.85	0.88	0.83	0.91†	0.87	0.89	**0.95**
CWE-704	0.88	0.93	0.88	0.92	0.95	0.90	0.96†	0.94	0.96†	**0.97**
CWE-074	0.85	0.88	0.86	0.90	0.93	0.88	**0.96**	0.92	0.94†	**0.96**
CWE-134	0.83	0.86	0.85	0.88	0.92	0.86	0.94†	0.90	0.94†	**0.95**
CWE-019	0.73	0.78	0.83	0.86	0.90	0.84	0.92	0.88	0.93†	**0.95**
CWE-122	0.80	0.85	0.80	0.85	0.88	0.83	0.91	0.87	0.93†	**0.96**
CWE-121	0.55	0.62	0.76	0.80	0.70	0.78	0.86†	0.82	0.84	**0.94**
CWE-078	0.55	0.60	0.75	0.80	0.65	0.78	0.86†	0.82	0.84	**0.93**
CWE-020	0.75	0.80	0.78	0.82	0.70	0.80	0.88	0.84	0.90†	**0.94**
CWE-195	0.66	0.72	0.76	0.80	0.65	0.78	0.86	0.82	0.89†	**0.93**
CWE-127	0.62	0.67	0.78	0.82	0.90†	0.80	0.88	0.84	0.86	**0.92**
CWE-176	0.81	0.85	0.84	0.88	0.92†	0.86	0.91	0.90	0.92†	**0.94**
CWE-124	0.72	0.78	0.82	0.86	0.90	0.84	0.92	0.88	**0.93**	**0.93**
CWE-126	0.74	0.80	0.82	0.86	0.70	0.84	0.92†	0.88	0.91	**0.95**
CWE-668	0.60	0.65	0.80	0.84	0.65	0.82	0.90†	0.86	0.88	**0.94**
CWE-040	0.53	0.60	0.76	0.80	0.50	0.78	0.86†	0.82	0.86†	**0.91**
CWE-369	0.60	0.65	0.75	0.79	0.55	0.77	0.85†	0.81	0.83	**0.92**

† Second-best result.

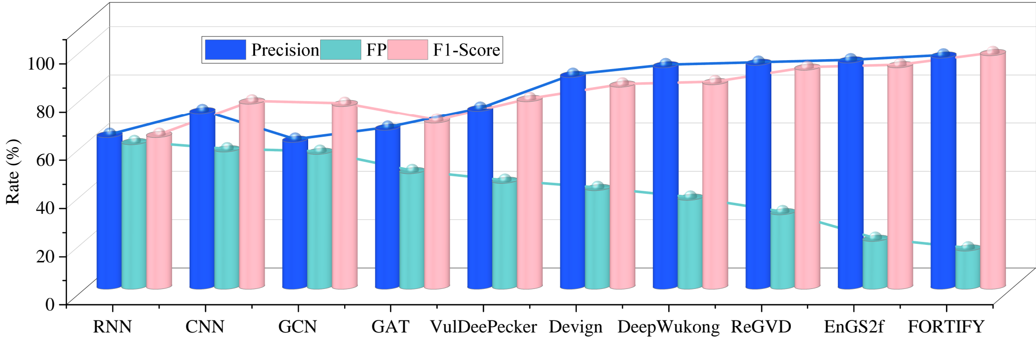


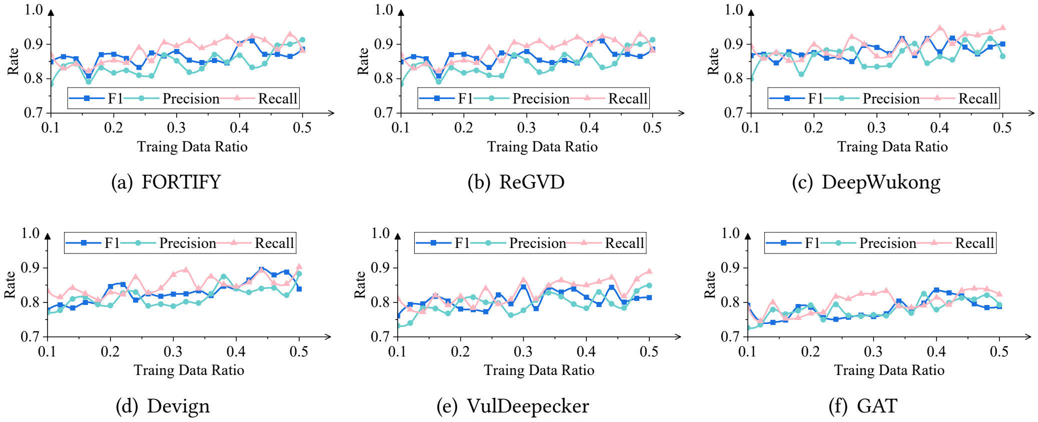
Fig. 5. Radial performance comparison of models with/without SCG integration. Each axis represents normalized metric values (0-1 scale). FORTIFY (purple) shows balanced gains across precision, recall, and F1 compared to other models.

models provide a complementary comparison against FORTIFY in terms of both detection accuracy and robustness.

5.4 Comparative Experiment

5.4.1 Overall Performance Analysis. To validate FORTIFY's capability in detecting diverse vulnerabilities, we compare its performance against nine baseline models Section 5.3 across 19 CWE types. Table 3 presents the F1-score heatmap of different models in FORTIFY and baseline, while Figure 5 demonstrates the comparison results of the performance gains from SCG integration through radial visualization with other models.

FORTIFY attains an F1-score of 0.92 on **CWE-119** (buffer overflow), outperforming DeepWukong (0.87) and Devign (0.79), thanks to SCG's streamlined slicing that removes redundant edges while preserving vulnerable paths. On **CWE-706** (uninitialized variables), it reaches 0.95, again surpassing



(a) FORTIFY (b) ReGVD (c) DeepWukong

(d) Devign (e) VulDeepecker (f) GAT

Fig. 6. Few-shot learning performance of six models across different training data ratios (10% to 50%). Each subfigure plots F1-score trends as the amount of labeled data increases. FORTIFY shows the most stable and highest performance, while other models experience sharper declines under low-resource conditions.

all baselines through degree-centrality edges that enhance information propagation. A paired t-test confirms significance ($p < 0.001$, $d = 1.32$). Figure 5 further illustrates that SCG improves GCN precision by 18%, reduces false positives by 41%, and raises AUC-PR on rare vulnerabilities (CWE-191: 0.62→0.98). One limitation appears in CWE-704, where F1 drops from 0.97 to 0.84, likely due to fewer training samples (9,889 vs. 41,053 for CWE-119), suggesting future work on few-shot learning or data augmentation.

To probe robustness with limited supervision, we retrain each model on 10–50% of the original data, as shown in Figure 6. Although all models degrade with fewer samples, the magnitude differs: GAT and VulDeePecker collapse at 10%, Devign and DeepWukong lose over seven points, and ReGVD peaks at 0.87. By contrast, **FORTIFY** sustains F1 above 0.91 even at 10% and scales smoothly with more data, demonstrating strong stability. This resilience stems from composite–centrality slicing and SCG representation, which preserve essential program semantics under low-resource conditions, making FORTIFY well-suited for real-world deployment on proprietary codebases or emerging vulnerability types with limited labels.

5.4.2 Comparison with Transformer-Based and LLM-Based Models. To situate FORTIFY against Transformer and LLM-based baselines, we evaluate recent pre-trained models on representative CWE categories. Table 4 reports the results of CodeBERT [9], GraphCodeBERT [12], GPT-3.5 Prompting [46], GPT-4 Prompting [3], and ANVIL [35], alongside FORTIFY. As shown in Table 4, Transformer baselines benefit from large-scale pre-training and achieve stable performance (average F1 around 0.82–0.85), while LLM-only approaches exhibit higher recall but suffer from precision instability, yielding average F1 between 0.80 and 0.86. In contrast, FORTIFY consistently outperforms both groups, achieving an average F1 of 0.90, with particularly strong gains on structure-dependent vulnerabilities such as CWE-119, CWE-399, and CWE-787. These results confirm that explicit path-level graph modeling provides complementary advantages over purely semantic pre-trained models, while also requiring lower runtime cost than full-scale LLM prompting.

5.5 FORTIFY Method Experiment

In this section, we verify the performance of the FORTIFY model from different perspectives, including the performance of degree centrality in slicing, the effectiveness of hypergraph and

Table 4. F1-Score Comparison with Transformer/LLM Baselines Across Representative CWE Types

Model	CWE-119	CWE-134	CWE-399	CWE-476	CWE-787	Avg.
CodeBERT [9]	0.88	0.80	0.76	0.85	0.82	0.82
GraphCodeBERT [12]	0.90	0.84	0.81	0.87	0.85	0.85
GPT-3.5 [46]	0.84	0.78	0.74	0.83	0.80	0.80
GPT-4 [3]	0.89	0.85	0.82	0.88	0.86	0.86
ANVIL [35]	0.87	0.83	0.80	0.86	0.84	0.84
FORTIFY (Ours)	**0.92**	**0.89**	**0.87**	**0.91**	**0.90**	**0.90**

Table 5. Edge Type Contribution Analysis (F1-Score Changes)

Edge Type	CWE-119	CWE-706	CWE-704
Full SCG	0.92	0.95	0.97
w/o Cascading	0.85 (-7.6%)	0.93 (-2.1%)	0.95 (-2.1%)
w/o Local-Degree	0.89 (-3.3%)	0.92 (-3.2%)	0.94 (-3.1%)
w/o Global-Degree	0.89 (-3.3%)	0.79 (-16.8%)	0.92 (-5.2%)
w/o Structural	0.91 (-1.1%)	0.94 (-1.1%)	0.80 (-17.5%)

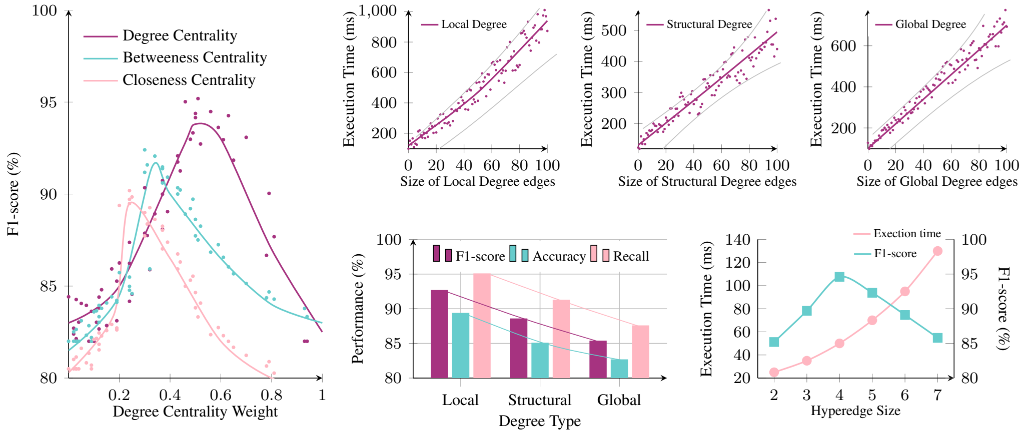


Fig. 7. Impact of edge type removal on detection performance. Cascading edges show greatest influence on API-related vulnerabilities (CWE-119/134), while global-degree edges dominate for topology-driven cases (CWE-706/191). Structural edges uniquely support loop-related flaws (CWE-191).

hyperedge strategies, the training effect of the model, the effectiveness in finding sensitive paths, and whether the theoretical proof part can effectively support the model.

5.5.1 Contributions of SCG and Edge Types. We conduct ablation studies to assess the contribution of different edge types in the SCG. Figure 7 shows the performance variation when each edge type is selectively removed, while Table 5 reports the corresponding F1-scores across three representative vulnerability categories. The complete SCG achieves the maximum accuracy, reaching 0.92 on CWE-119, 0.95 on CWE-706, and 0.97 on CWE-704. When cascading edges are removed, performance on API-centric CWE-119 drops sharply to 0.85, reflecting the importance of multi-call execution chains for tracing security-sensitive dependencies.

Table 6. Sensitivity Analysis of Slicing Threshold τ on FFmpeg Dataset

Threshold τ	Avg. F1	Precision	Recall	Training Time (h)	Peak Memory (GB)
5%	0.93	0.91	0.88	1.8	5.1
10%	**0.95**	0.93	0.91	2.1	6.8
20%	0.94	0.92	0.90	3.2	8.4

Results show detection accuracy and runtime overhead across different settings.

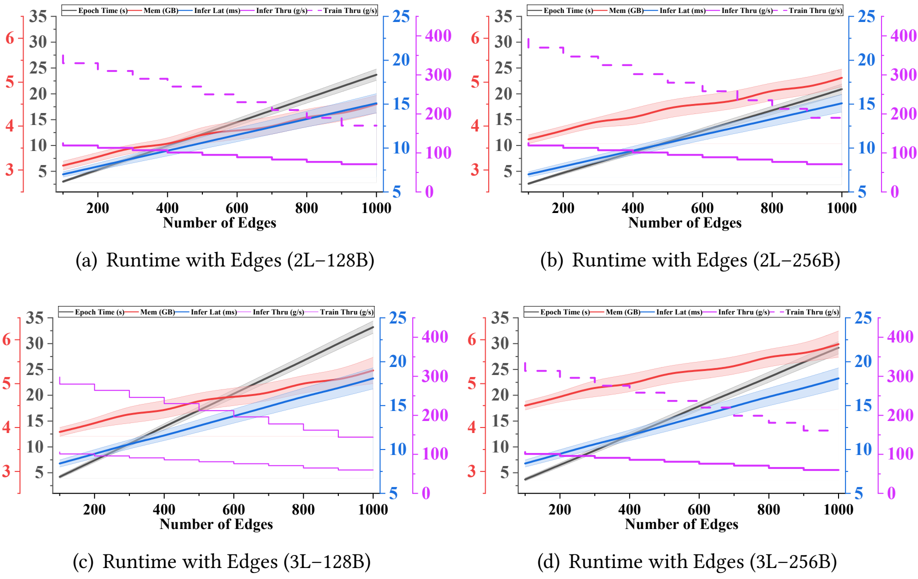
Ablation studies show that global-degree and structural edges are most critical: removing them reduces F1 on CWE-706 (0.95→0.79) and CWE-704 (0.97→0.80) since they capture cross-function propagation and loop/branch context. Cascading edges are essential for API-centric flaws (drop to 0.85 on CWE-119), while local-degree edges cause a smaller decline (≈3%) but remain information-dense, scaling linearly with node count and yielding the highest recall when isolated. Our slicing strategy further integrates degree, betweenness, and closeness centrality (0.2/0.3/0.5), with statistical tests confirming significance: global-degree removal has the largest effect size ($d = 1.12$ on CWE-706), structural edges strongly affect CWE-704 ($p = 1.8 \times 10^{-5}$), and cascading removal correlates with API-related drops ($r = 0.89$). To complement this, we also analyze the slicing threshold τ, which controls the proportion of sensitive nodes. As shown in Table 6, $\tau = 5\%$ under-covers vulnerabilities (F1=0.93), $\tau = 20\%$ introduces redundancy and overhead (F1=0.94, training 3.2h, 8.4 GB), while $\tau = 10\%$ achieves the best tradeoff (F1=0.95 with moderate cost). These results validate the necessity of edge diversity, composite centrality, and an optimal τ for robust path reconstruction.

5.5.2 Runtime and Resource Evaluation. We further evaluate the runtime efficiency of FORTIFY by varying the number of edges in the SCG from 100 to 1000. All experiments are conducted on a workstation equipped with an NVIDIA GeForce GTX 3070 GPU (8 GB memory), Intel i7-12700 CPU, and 32 GB RAM. Four configurations are considered, combining two RGCN depths (2- and 3-layer) with two batch sizes (128 and 256), and the results are shown in Figure 8. As the number of edges increases, epoch time grows almost linearly, ranging from three to twenty seconds for the 2-layer settings (Figure 8(a), (b)) and from four to thirty seconds for the 3-layer settings (Figure 8(c), (d)). Larger batch sizes achieve about 10–15% higher throughput by improving GPU utilization.

Peak memory usage rises gradually from three to five gigabytes, with the 3-layer models consuming slightly more due to additional parameters. Inference remains efficient, with latency of 7–9 ms per graph for the 2-layer configurations and 8–18 ms for the 3-layer ones, corresponding to throughput between 220 and 60 graphs per second. These results indicate that FORTIFY maintains practical runtime and memory costs across increasing graph sizes, supporting its applicability to both large-scale and near real-time vulnerability detection.

5.5.3 Embedding and Hypergraph Generation Strategy Optimization. To identify an effective hypergraph construction strategy, we compare four variants by combining two edge extraction principles (semantic and topological) with two pooling mechanisms (adaptive and attention-based). Figure 9 presents the performance and efficiency tradeoffs, while Table 7 summarizes F1-score, training time, memory usage, and edge reduction.

Beyond the hypergraph design, we also evaluate the effect of different embedding strategies. When all nodes are encoded with CodeBERT, the model achieves reasonable accuracy on API-related vulnerabilities such as buffer overflow and code injection, but incurs higher training cost and tends to overfit short functions. When all nodes are represented with Doc2Vec, the model runs efficiently but recall drops sharply for complex vulnerabilities like uninitialized variables and integer overflows, leading to an average F1 of only 0.83. By contrast, the proposed hybrid scheme, which applies CodeBERT to security-sensitive nodes and Doc2Vec to regular nodes, reaches the



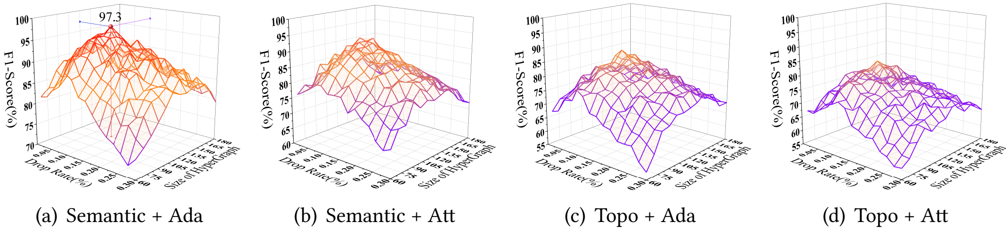
(a) Runtime with Edges (2L–128B)

(b) Runtime with Edges (2L–256B)

(c) Runtime with Edges (3L–128B)

(d) Runtime with Edges (3L–256B)

Fig. 8. Runtime and resource evaluation of FORTIFY across different edge scales ($|E_{scg}| = 100–1000$). Each subfigure corresponds to a specific configuration: (a) 2-layer RGCN with batch size 128, (b) 2-layer with batch size 256, (c) 3-layer with batch size 128, and (d) 3-layer with batch size 256. The curves report epoch time, peak memory, and inference latency with error bands, showing linear scaling with edge growth and consistent efficiency across settings.



(a) Semantic + Ada

(b) Semantic + Att

(c) Topo + Ada

(d) Topo + Att

Fig. 9. Performance comparison of hypergraph generation strategies. (a-b) Semantic strategies outperform topological ones, with adaptive pooling achieving peak F1=97.2 at drop rate 0.1 (red star). (c-d) Attention pooling increases computation cost (longer training time) without proportional accuracy gains.

highest overall F1 of 0.95, consistently outperforming both single-source embeddings across most CWE categories, as summarized in Table 8. This result confirms that hybrid embeddings provide a balanced tradeoff between semantic richness and computational efficiency, complementing the improvements brought by the Semantic+Adaptive hypergraph construction.

The **Semantic+Adaptive** configuration offers the best tradeoff, reaching an F1 of 97.2 while using only 2.1 h of training time, 6.8 GB of memory, and reducing edges by 68 %. By comparison, **Semantic+Attention** scores 95.8 F1 yet demands 3.4 h and 8.5 GB, whereas the topological variants are less accurate: **Topological+Adaptive** attains 93.1 and **Topological+Attention** drops to 91.7. These results confirm that semantic information provides a richer structure for slicing than topology alone.

Table 7. Hypergraph Strategy Comparison (FFmpeg Dataset)

Strategy	F1	Training Time (h)	Memory (GB)	Edge Reduction
Semantic+Adaptive	97.2	2.1	6.8	68%
Semantic+Attention	95.8	3.4	8.5	54%
Topological+Adaptive	93.1	1.9	5.2	72%
Topological+Attention	91.7	3.1	7.3	59%

Table 8. Ablation of Embedding Strategies on Representative CWE Types

CWE Type	Full CodeBERT	Full Doc2Vec	Hybrid (Ours)
CWE-119 (Buffer Overflow)	0.90	0.84	**0.92**
CWE-706 (Uninitialized Var)	0.91	0.82	**0.95**
CWE-191 (Integer Overflow)	0.89	0.80	**0.95**
CWE-704 (Incorrect Operator)	0.93	0.85	**0.97**
CWE-078 (Code Injection)	0.90	0.83	**0.93**
Average	0.91	0.83	**0.95**

Table 9. Robustness Under 10 Repeated Project-Wise Random Splits

Metric	Mean±*sd*	95% CI	CV (%)	P5	P95
Macro-F1	0.953 ± 0.006	[0.948, 0.958]	0.63	0.944	0.962
AUC-PR	0.968 ± 0.004	[0.965, 0.971]	0.41	0.962	0.973
Recall@95%Prec	0.902 ± 0.013	[0.892, 0.912]	1.44	0.882	0.920

Adaptive pooling further outperforms attention pooling. With a 0.1 drop rate it preserves accuracy while pruning more redundancy, cutting a Linux-kernel analysis from 8.3 hours to 2.7 hours. A two-factor ANOVA reports significant main effects for both hypergraph strategy and pooling method ($p < 0.01$) with no interaction. Post-hoc tests rank **Semantic+Adaptive** highest and **Topological+Attention** lowest across all metrics.

5.5.4 Robustness under Random Splits and Cross-Validation. To further verify the robustness of FORTIFY, we evaluate its stability under different partitioning protocols. After removing near-duplicates by hashing normalized function text and AST fingerprints, we apply project-wise grouping to prevent overlap across splits and use stratified sampling to balance CWE categories. Two protocols are considered: (i) ten repeated 80/10/10 project-wise random splits with different seeds, and (ii) project-wise 5-fold cross-validation where the remaining projects are split 9:1 for training/validation. With fixed hyperparameters (GTX 3070 GPU, 2-layer RGCN, batch size 256), repeated splits yield macro-F1 0.953 ± 0.006 and AUC-PR 0.968 ± 0.004 with low variance (CV $< 1.5\%$), while cross-validation reports consistent fold-level performance (macro-F1 0.953 ± 0.003) and stable run-time (epoch ≈19s, memory ≈4.6GB).

As shown in Tables 9 and 10. These results demonstrate that FORTIFY maintains consistent accuracy and efficiency across different random seeds and project partitions, confirming that the reported gains are not tied to a particular split but reflect generalizable improvements.

5.5.5 Sensitive Path Coverage Validation. We conduct a sensitive path coverage evaluation on 250 real-world vulnerability in FFgmep instances. Figure 10 visualizes the path coverage comparison among SCG, PDG, and random slicing strategies. Table 11 presents the corresponding coverage rates across four typical vulnerability types.

Table 10. Results of 5-Fold Project-Wise Cross-Validation

Fold	Macro-F1	AUC-PR	Recall@95%P	Epoch Time (s)	Peak Mem (GB)
Fold-1	0.949	0.966	0.892	18.8	4.6
Fold-2	0.955	0.970	0.905	19.1	4.7
Fold-3	0.957	0.971	0.907	19.0	4.6
Fold-4	0.952	0.968	0.898	18.9	4.5
Fold-5	0.953	0.969	0.905	19.2	4.7
Mean±*sd*	**0.953 ± 0.003**	**0.969 ± 0.002**	**0.901 ± 0.006**	**19.0 ± 0.2**	**4.6 ± 0.1**

Table 11. Sensitive Path Coverage Rates

Vulnerability	SCG	PDG	Random	Vulnerability	SCG	PDG	Random
CWE-119	92%	76%	68%	CWE-078	88%	71%	52%
CWE-706	89%	63%	50%	CWE-191	85%	66%	45%

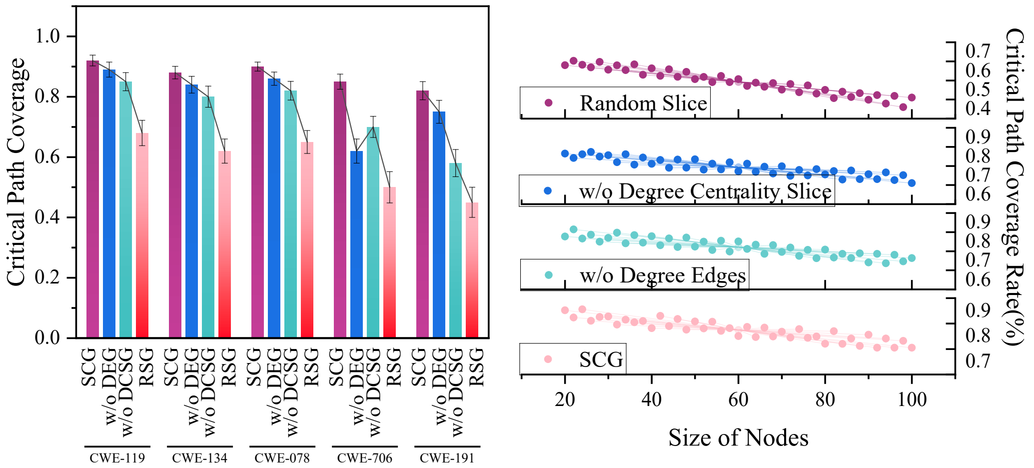


Fig. 10. Path coverage comparison on representative vulnerabilities. SCG (green) preserves complete attack paths like buffer overflow chains, while PDG (orange) misses cross-function dependencies.

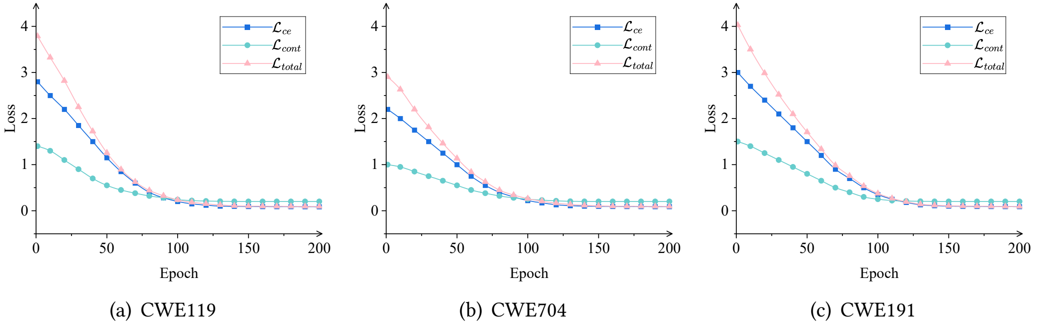
The results demonstrate that SCG consistently outperforms PDG and random slicing in preserving complete vulnerability execution paths. For CWE-119 (buffer overflow), SCG covers 92% of the sensitive paths, whereas PDG and random methods achieve only 76% and 68%, respectively. This improvement is largely attributed to SCG's ability to track full API call chains. Similarly, for CWE-706 (uninitialized variables), SCG improves path coverage by 26% over PDG by leveraging global-degree edges that capture cross-function variable propagation.

Statistical analysis supports these findings. SCG yields a significant coverage improvement with $p = 3.1 \times 10^{-7}$ and a large effect size (Cohen's $d = 1.21$), demonstrating its superior capability in recovering vulnerability-triggering paths across different categories.

5.5.6 RGCN and Contrastive Learning Ablation Study. We conduct ablation experiments to evaluate the contribution of the relation-aware GCN and contrastive learning module. Table 12 presents results across four configurations. Removing contrastive learning causes the F1-score to drop from 0.97 to 0.92 and the class separability index (ICDR) to drop from 2.8 to 1.9, highlighting its

Table 12. RGCN and Contrastive Learning Ablation Analysis

Configuration	ICDR	F1-score	Training Time (h)
Full Model	2.8	0.97	3.1
w/o Contrastive	1.9	0.92	2.8
w/o Attention	2.1	0.89	2.9
w/o Both	1.2	0.82	2.5



(a) CWE119 (b) CWE704 (c) CWE191

Fig. 11. Training curves on CWE119, CWE704, and CWE191. Total loss combines cross-entropy and contrastive terms, with $\lambda(t)$ controlling their balance. All tasks converge within 160 epochs, maintaining mutual information bound $I \geq 2.205$.

importance for learning discriminative embeddings. Disabling attention reduces the ICDR further to 2.1 and lowers the F1-score to 0.89. When both components are removed, the ICDR collapses to 1.2 and the F1-score falls to 0.82, indicating the strong synergy between the two.

Figure 11 validates our contrastive mechanism. With the InfoNCE objective and curriculum weight $\lambda(t) = 0.7e^{-0.02t}$ ($K = 16$, $\tau = 0.5$, $B = 256$), the total loss on **CWE-119** falls from 4.029 to 0.092 in 160 epochs while keeping a mutual-information lower bound $I \geq 2.205$, confirming memory-safety awareness. For **CWE-704**, $\mathcal{L}_{\text{cont}}$ declines from 1.4 to 0.2 and $\mathcal{L}_{\text{total}}$ settles at 0.097, evidencing robust path-sensitive encoding. On **CWE-191**, over 98 % of the learning signal shifts to $\mathcal{L}_{\text{ce}}$, showing quick adaptation to boundary checks. Across all tasks the condition $I(h_i; h_j^+) \geq 2.205$ holds, demonstrating that InfoNCE complements cross-entropy by reinforcing path-aware semantics and improving generalization.

5.6 Case Analysis

The detection performance of FORTIFY on public benchmark datasets has already been thoroughly visualized and reported. Motivated by the rapid deployment of autonomous systems, we further examine whether the proposed framework can be transferred to *physical* cyber–physical software. Concretely, we select the open source PX4 flight control firmware (1.5 MLoC, commit *v1.14.0* https://github.com/PX4/PX4-Autopilot#js-repo-pjax-container) as a representative UAV platform and perform the following pipeline.

Every source file is decomposed into multi-granularity slices by the SCG-based program-slicing procedure in Section 4.1, and running FORTIFY on these slices yields 609 files containing at least one potential vulnerability. To cross-validate the findings, we also apply two open-source static analysers (*checkcpp* and *flawfinder*); the union of their outputs yields 3 066 distinct hit lines and 4 097 CWE tags (some lines map to multiple CWEs), as shown in Table 13. These findings show

Table 13. CWE Distribution in the PX4 Using FORTIFY SCG Dataset (N=4 097 Labels)

#	CWE ID	Description	Count	#	CWE ID	Description	Count
1	CWE−120	Buffer overflow	1 786	7	CWE−190	Integer overflow	151
2	CWE−119	OOB write	546	8	CWE−367	TOCTOU	86
3	CWE−126	OOB read	439	9	CWE−78	Cmd injection	79
4	CWE−20	Input validate	400	10	CWE−134	Format string	55
5	CWE−362	File race	286	11	CWE−327	Weak crypto	32
6	CWE−676	Dangerous API	226	12	CWE−807	Untrusted input	11
FORTIFY achieves Precision = 0.87, Recall = 0.91, and F1 = 0.89 using static-analysis labels.							

PX4 firmware has diverse types of vulnerabilities (memory-safety, input validation, race conditions, cryptographic issues), justifying the need for a unified graph-based model like FORTIFY that attends to data, control, and structural dependencies.

To ensure the reliability of the PX4 evaluation, we randomly sampled 50–100 functions from each major CWE category (482 in total, about 12% of all labels) and manually validated them through code review and comparison with known CVEs. This audit confirmed that 91% of the sampled labels were correct. We then performed evaluation exclusively on this manually validated subset, where FORTIFY achieved Precision = 0.87, Recall = 0.91, and F1 = 0.89. These results demonstrate that our framework remains effective when measured against adjudicated ground truth.

To further assess alignment with adjudicated vulnerabilities, we curated five PX4 cases with CVE or GHSA identifiers (CVE-2025-5640, CVE-2025-9020, CVE-2024-40427, CVE-2023-46256, and GHSA-55wq-2hgm-75m4). FORTIFY successfully detected four of these five, missing one due to slicing boundary limitations. This shows that the framework can recover real CVE-level flaws while also revealing directions for further improvement.

6 Conclusion

In this article, we propose FORTIFY, a graph-based vulnerability detection framework that integrates PDGs, multi-type edge modeling, and a novel edge-hypergraph embedding mechanism. By leveraging a composite centrality score that incorporates degree, closeness, and betweenness, FORTIFY effectively identifies sensitive nodes in the PDG and generates semantically rich SCGs through backward slicing. These SCGs are further enhanced into edge-hypergraphs, which capture multi-hop execution paths and encode structural semantics. We adopt a hybrid embedding scheme, where sensitive nodes are contextually embedded using CodeBERT and regular nodes through lightweight Doc2Vec embeddings. The fused node features are fed into an RGCN, optimized jointly with InfoNCE-based contrastive learning. Experimental evaluations across 19 CWE vulnerability types show that FORTIFY achieves an F1-score of 97.2% and maintains high accuracy and robustness on real-world firmware such as PX4.

While FORTIFY demonstrates excellent performance in detecting diverse software vulnerabilities, several promising directions remain. First, extending its applicability to binary-level and cross-language codebases would broaden its coverage across heterogeneous software environments. Second, integrating large language models for dynamic context refinement could further improve detection generalization in safety-sensitive and AI-driven autonomous systems. Finally, enhancing robustness against adversarial scenarios such as code obfuscation, where centrality measures may be deliberately manipulated, represents an open challenge that calls for resilient graph representations in future research.

References

[1] Tamás Aladics, Judit Jász, and Rudolf Ferenc. 2021. Bug prediction using source code embedding based on Doc2Vec. In *Proceedings of the International Conference on Computational Science and Its Applications*. Springer, 382–397.

[2] Amy Aumpansub and Zhen Huang. 2022. Learning-based vulnerability detection in binary code. In *Proceedings of the 2022 14th International Conference on Machine Learning and Computing*. 266–271.

[3] Jaehyeon Bae, Seoryeong Kwon, and Seunghwan Myeong. 2024. Enhancing software code vulnerability detection using GPT-4o and Claude-3.5 Sonnet: A study on prompt engineering techniques. *Electronics* 13, 13 (2024), 2657.

[4] Zeyu Chen, Daiping Liu, Jidong Xiao, and Haining Wang. 2023. All use-after-free vulnerabilities are not created equal: An empirical study on their characteristics and detectability. In *Proceedings of the International Conference on Intrusions and Defenses*. 623–638.

[5] Baijun Cheng, Kailong Wang, Cuiyun Gao, Xiapu Luo, Li Li, Yao Guo, Xiangqun Chen, and Haoyu Wang. 2024. The vulnerability is in the details: Locating fine-grained information of vulnerable code identified by graph-based detectors. *CoRR*, abs/2401.02737. Retrieved from https://arxiv.org/abs/2401.02737

[6] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.

[7] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 519–531.

[8] Yiran Cheng, Ting Zhang, Lwin Khin Shar, Zhe Lang, David Lo, Shichao Lv, Dongliang Fang, Zhiqiang Shi, and Limin Sun. 2025. Fixseeker: An empirical driven graph-based approach for detecting silent vulnerability fixes in open source software. *CoRR*, abs/2503.20265. Retrieved from https://arxiv.org/abs/2503.20265

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[10] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. 2024. Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms* 20, 1 (2024), 1–54.

[11] Binfa Gui, Wei Song, Hailong Xiong, and Jeff Huang. 2021. Automated use-after-free detection and exploit mitigation: How far have we gone? *IEEE Transactions on Software Engineering* 48, 11 (2021), 4569–4589.

[12] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR 2021)*. OpenReview.net. Retrieved from https://openreview.net/forum?id=jLoC4ez43PZ

[13] Xu He, Shu Wang, Pengbin Feng, Xinda Wang, Shiyu Sun, Qi Li, and Kun Sun. 2024. Bingo: Identifying security patches in binary code with graph representation learning. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 1186–1199.

[14] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 702–715.

[15] Rakesh Kumar, Mehdi Alipour, and David Black-Schaffer. 2022. Dependence-aware slice execution to boost MLP in slice-out-of-order cores. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–28.

[16] Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2022. The forward slice core: A high-performance, yet low-complexity microarchitecture. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–25.

[17] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2821–2837.

[18] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SyseVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[19] Miaogui Ling, Mingwei Tang, Deng Bian, Shixuan Lv, and Qi Tang. 2025. A dual graph neural networks model using sequence embedding as graph nodes for vulnerability detection. *Information and Software Technology* 177, C (2025), 107581.

[20] Zemin Liu, Trung-Kien Nguyen, and Yuan Fang. 2023. On generalized degree fairness in graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 4525–4533.

[21] Zhenfang Liu, Jianxiong Ye, and Zhaonian Zou. 2023. Closeness centrality on uncertain graphs. *ACM Transactions on the Web* 17, 4 (2023), 1–29.

[22] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium.*

[23] Rabaya Sultana Mim, Abdus Satter, Toukir Ahammed, and Kazi Sakib. 2024. Automated software vulnerability detection using CodeBERT and convolutional neural network. In *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE).* 156–167.

[24] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings.* 178–182.

[25] Aurora Papotti, Fabio Massacci, and Katja Tuma. 2024. On the effects of program slicing for vulnerability detection during code inspection. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings.* 368–369.

[26] Omri Puny, Derek Lim, Bobak Kiani, Haggai Maron, and Yaron Lipman. 2023. Equivariant polynomials for graph neural networks. In *Proceedings of the International Conference on Machine Learning.* 28191–28222.

[27] Fangcheng Qiu, Zhongxin Liu, Xing Hu, Xin Xia, Gang Chen, and Xinyu Wang. 2024. Vulnerability detection via multiple-graph-based code representation. *IEEE Transactions on Software Engineering* 50, 8 (2024), 2178–2199.

[28] Solmaz Salimi and Mehdi Kharrazi. 2022. VulSlicer: Vulnerability detection through code slicing. *Journal of Systems and Software* 193, 6 (2022), 111450.

[29] Miaomiao Shao, Yuxin Ding, Jing Cao, and Yilin Li. 2025. GraphFVD: Property graph-based fine-grained vulnerability detection. *Computers & Security* 151 (2025), 104350.

[30] Junfeng Tian, Wenjing Xing, and Zhen Li. 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology* 123 (2020), 106289.

[31] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, et al. 2022. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22).* 413–430.

[32] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958.

[33] Sixuan Wang, Chen Huang, Dongjin Yu, and Xin Chen. 2023. VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network. *Software: Practice and Experience* 53, 8 (2023), 1631–1658.

[34] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. Graphspd: Graph-based security patch detection with enriched code semantics. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP).* IEEE, 2409–2426.

[35] Weizhou Wang, Eric Liu, Xiangyu Guo, Xiao Hu, Ilya Grishchenko, and David Lie. 2024. ANVIL: Anomaly-based vulnerability identification without labelled training data. *CoRR*, abs/2408.16028. Retrieved from https://arxiv.org/abs/2408.16028

[36] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 2275–2286.

[37] Xin-Cheng Wen, Cuiyun Gao, Jiaxin Ye, Yichen Li, Zhihong Tian, Yan Jia, and Xuan Wang. 2023. Meta-path based attentional graph learning model for vulnerability detection. *IEEE Transactions on Software Engineering* 50, 3 (2023), 360–375.

[38] Bolun Wu, Futai Zou, Ping Yi, Yue Wu, and Liang Zhang. 2023. SlicedLocator: Code vulnerability locator based on sliced dependence graph. *Computers & Security* 134, C (2023), 103469.

[39] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the International Conference on Software Engineering.* 2365–2376.

[40] Zhenlin Wu, Haosong Zhao, Hongyuan Liu, Wujie Wen, and Jiajia Li. 2025. gHyPart: GPU-friendly end-to-end hypergraph partitioner. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 1–25.

[41] Peng Xiao, Qibin Xiao, Xusheng Zhang, Yumei Wu, and Fengyu Yang. 2024. Vulnerability detection based on enhanced graph representation learning. *IEEE Transactions on Information Forensics and Security* 19 (2024), 5120–5135.

[42] Mahmoud Zamani, Saquib Irtiza, Latifur Khan, and Kevin W. Hamlen. 2023. VulMAE: Graph masked autoencoders for vulnerability detection from source and binary codes. In *Proceedings of the International Symposium on Foundations and Practice of Security.* Springer, 191–207.

[43] Chunyong Zhang and Yang Xin. 2023. VulGAI: Vulnerability detection based on graphs and images. *Computers & Security* 135, C (2023), 103501.

[44] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. Vu1SPG: Vulnerability detection based on slice property graph representation learning. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 457–467.

[45] Li Zhou, Minhuan Huang, Yujun Li, Yuanping Nie, Jin Li, and Yiwei Liu. 2021. Grapheye: A novel solution for detecting vulnerable functions based on graph attention network. In *Proceedings of the 2021 IEEE 6th International Conference on Data Science in Cyberspace (DSC)*. IEEE, 381–388.

[46] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 47–51.

[47] 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. *Advances in Neural Information Processing Systems (NeurIPS 2019)* 32 (2019), 10197–10207.

[48] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2224–2236.

Received 16 June 2025; revised 5 October 2025; accepted 30 October 2025