# VISION: Robust and Interpretable Code Vulnerability Detection Leveraging Counterfactual Augmentation

**David Egea**[1 2], **Barproda Halder**[1], **Sanghamitra Dutta**[1]
[1]University of Maryland College Park
[2]Universidad Pontificia Comillas
davidegea@alu.comillas.edu, bhalder@umd.edu, sanghamd@umd.edu

## Abstract

Automated detection of vulnerabilities in source code is an essential cybersecurity challenge, underpinning trust in digital systems and services. Graph Neural Networks (GNNs) have emerged as a promising approach as they can learn the structural and logical code relationships in a data-driven manner. However, the performance of GNNs is severely limited by training data imbalances and label noise. GNNs can often learn "spurious" correlations due to superficial code similarities in the training data, leading to detectors that do not generalize well to unseen real-world data. In this work, we propose a new unified framework for robust and interpretable vulnerability detection—that we call VISION—to mitigate spurious correlations by systematically augmenting a counterfactual training dataset. Counterfactuals are samples with minimal semantic modifications that have opposite prediction labels. Our complete framework includes: (i) generating effective counterfactuals by prompting a Large Language Model (LLM); (ii) targeted GNN model training on synthetically paired code examples with opposite labels; and (iii) graph-based interpretability to identify the truly crucial code statements relevant for vulnerability predictions while ignoring the spurious ones. We find that our framework reduces spurious learning and enables more robust and generalizable vulnerability detection, as demonstrated by improvements in overall accuracy (from 51.8% to 97.8%), pairwise contrast accuracy (from 4.5% to 95.8%), and worst-group accuracy increasing (from 0.7% to 85.5%) on the widely popular Common Weakness Enumeration (CWE)-20 vulnerability. We also demonstrate improvements using our proposed metrics, namely, intra-class attribution variance, inter-class attribution distance, and node score dependency. We provide a new benchmark for vulnerability detection, CWE-20-CFA, comprising 27,556 samples from functions affected by the high-impact and frequently occurring CWE-20 vulnerability, including both real and counterfactual examples. Furthermore, our approach enhances societal objectives of transparent and trustworthy AI-based cybersecurity systems through interactive visualization for human-in-the-loop analysis.

## 1 Introduction

Software vulnerabilities remain a primary entry point for cyberattacks, making early and accurate detection essential for securing modern digital systems (Chernis and Verma 2018). The deep syntactic and semantic structure of code remains difficult for traditional static and dynamic analysis tools to model, restricting their applicability and effectiveness (Yamaguchi et al. 2014). To this end, Graph Neural Networks (GNNs) (Scarselli et al. 2009; Wang et al. 2023) offer a promising approach for vulnerability detection by representing source code as graphs that capture syntax, control flow, and data dependencies. Architectures such as Devign (Zhou et al. 2019) exemplify the potential of GNN-based methods to automatically learn complex program semantics and support data-driven vulnerability detection.

However, the effectiveness of GNN-based detectors is constrained by the quality of the datasets they are trained on. Popular benchmarks often suffer from label duplication, class imbalance, and inconsistent or noisy labeling (Ding et al. 2024; Guo and Bettaieb 2023; Croft, Babar, and Kholoosi 2023), which can lead to spurious correlations. Spurious correlations are formally defined as statistical associations between input features and target labels that do not reflect a true causal relationship; instead, they arise from coincidental patterns or confounding variables in the data (Haig 2003; Ye et al. 2024; Bell and Wang 2024; Steinmann et al. 2024; Chen et al. 2023a; Halder et al. 2024). Models influenced by such correlations may appear accurate during training but often fail to generalize, as they rely on misleading or dataset-specific signals. These limitations are further compounded by the fact that GNN-based detectors behave as opaque boxes, offering limited interpretability and transparency into their decision-making process. Simply predicting vulnerabilities is not enough, as it remains unclear *what aspects of the input does the model rely on and if those aspects are really crucial*. Without this understanding, it becomes difficult to ensure robust and trustworthy decisions.

To address these limitations, we propose **VISION** (**V**ulnerability **I**dentification and **S**puriousness mitigation via counterfactual augmentat**ION**), a unified framework for robust and interpretable vulnerability detection by systematically augmenting a counterfactual dataset. In our context, we define *counterfactuals* as minimally altered code examples whose vulnerability labels are inverted (e.g., from vulnerable to benign or vice versa), inspired from counterfactual explanations in tabular classification (Wachter, Mittelstadt, and Russell 2018). As a simple illustration, consider the transformation of a

line like `strcpy(dest, "fixed_string");` into `strcpy(dest, user_input);`. While syntactically similar, the latter introduces a potential vulnerability by incorporating unvalidated input. *By generating such minimally modified examples during training, our framework encourages GNN models to better distinguish genuine vulnerability patterns from spurious ones.* To support this, we train a GNN model inspired by the Devign architecture (Zhou et al. 2019) on these counterfactual pairs, and then integrate the Illuminati explainer (He, Ji, and Huang 2023) to identify *the most influential code statements* via subgraph-based attributions.

We evaluate our framework on a challenging large-scale vulnerability detection dataset called the Common Weakness Enumeration (CWE)-20 (MITRE 2025), consisting of Improper Input Validation—a high-impact vulnerability ranked 4th in the 2022 CWE Top 25—highlighting our potential to enhance robustness and generalization in critical security contexts. We find that counterfactual augmentation leads to substantial improvements across key robustness and generalization metrics—including pairwise accuracy (rising from 4.5% to 95.8%) and worst-group accuracy (from 0.7% to over 85%)—demonstrating our effectiveness in mitigating spurious correlations and enhancing model generalization.

Our framework also includes an interactive visualization module for attributing the most influential statements in the source code (explainable inspection) for human-in-the-loop settings. Explanations generated under our framework tend to prioritize semantically relevant code regions, suggesting reduced reliance on spurious patterns. In essence, our framework VISION is a significant advancement in explainability for vulnerability detection, that can potentially help practitioners understand why code is flagged as risky, enabling more reliable remediation (Sharma et al. 2022).

To summarize, our key contributions are:

- **Novel counterfactual data augmentation strategy** leveraging Large Language Models (LLMs) to improve the robustness of GNN-based vulnerability detection by effectively mitigating spurious correlations.
- **Empirical validation** on a challenging vulnerability detection dataset called CWE-20, demonstrating substantial improvements on several generalization metrics, showcasing a scalable way to improve performance without relying on new data sources.
- **Extensive benchmark** that we call CWE-20-CFA constructed by generating counterfactual examples from existing CWE-20 samples.
- **Visualization module** for qualitative analysis of the model's decision-making, showing more semantically-meaningful input source code attributions.

## 2 Related Work

**Vulnerability Detection with Machine Learning.** Early machine learning methods used handcrafted features like token frequencies and structural cues but struggled to generalize across codebases (Neuhaus et al. 2007; Chernis and Verma 2018). Deep learning improved detection leveraging RNNs for sequential dependencies (Ziems and Wu 2021), CNNs for local patterns (Wu et al. 2022), and transformers like CodeBERT and GraphCodeBERT for long-range context (Feng et al. 2020; Guo et al. 2021). GNNs model source code as structured graphs (Scarselli et al. 2009; Zhou et al. 2019; Yamaguchi et al. 2014), thus enhancing robustness. LLMs have also been directly applied for cross-language and general-purpose vulnerability detection (Sultana, Afreen, and Eisty 2024; Zhou, Zhang, and Lo 2024).

**Dataset Quality and Challenges.** Label noise is a critical problem, with mislabeling rates reported between 20–71% in several benchmark datasets (Croft, Babar, and Kholoosi 2023). Duplication rates from 17–99% can cause overfitting to surface-level patterns, while class imbalance biases models toward non-vulnerable classes (Guo and Bettaieb 2023). Sample similarity—where inter-class examples are too alike, or intra-class examples are too diverse—further complicates learning (Liu et al. 2022b). Recent datasets like PrimeVul (Ding et al. 2024) introduce stricter de-duplication and validation. Others, such as DiverseVul (Chen et al. 2023b), aim to increase linguistic and structural diversity, improving generalization. Yet high-quality, well-balanced vulnerability datasets remain scarce, posing a significant barrier to developing robust detection models.

**Data Augmentation Strategies for Source Code.** Augmentation methods for vulnerability detection aim to preserve both syntactic validity and semantic meaning. Techniques include CodeGraphSMOTE (Ganz et al. 2023), which interpolates in graph latent space to balance datasets, and transformation-based methods that diversify code while preserving labels (Liu et al. 2024). LLM-based approaches like VulScribeR (Daneshvar et al. 2024) offer scalable synthesis of vulnerable code. Outside of the source code domain, other related works (Kaushik, Hovy, and Lipton 2020; Ross, Marasović, and Peters 2021; Dissanayake and Dutta 2024; Temraz and Keane 2021; Ding et al. 2022; Kong et al. 2022; Liu et al. 2022a) have explored strategies for augmenting synthetic examples in the dataset, e.g., finding counterfactuals via optimization for numeric data, manually creating new examples for text data, oversampling underrepresented groups, or perturbation-based techniques. However, counterfactual augmentation for code vulnerability detection—focusing on minimal, label-flipping edits leveraging an LLM—remains largely unexplored.

**Explainability in Vulnerability Detection.** Model-agnostic tools like SHAP and LIME explain feature contributions (Lundberg and Lee 2017; Ribeiro, Singh, and Guestrin 2016). In code analysis, IVDetect (Li, Wang, and Nguyen 2021) leverages program dependency graphs to localize vulnerabilities. CFExplainer (Chu et al. 2024) generates counterfactuals to highlight decision boundaries. GNNExplainer and PGM-Explainer provide subgraph-based explanations for GNNs (Ying et al. 2019; Vu and Thai 2020). Illuminati (He, Ji, and Huang 2023) is a domain-specific explainer for GNNs in cybersecurity that identifies the most influential nodes, edges, and attributes in a prediction, producing interpretable subgraphs that clarify the model's decision-making process. Notably, Illuminati

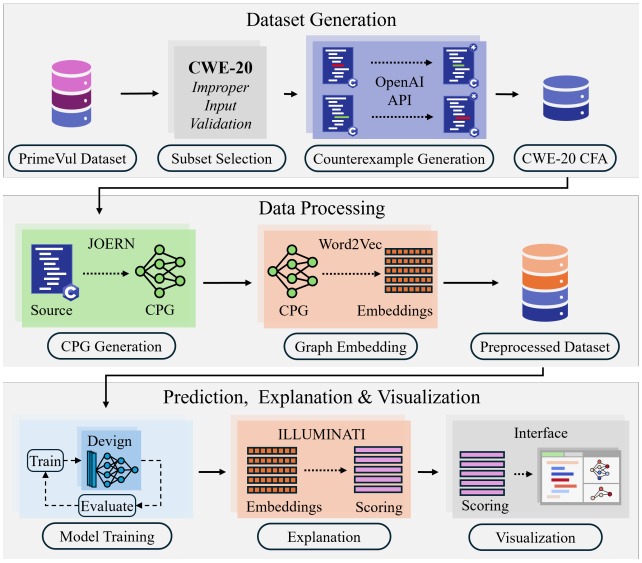


Figure 1: Complete architecture of our VISION Framework. Starting from the original PrimeVul dataset, the framework illustrates the end-to-end process: dataset filtering to CWE-20, counterfactual generation and class balancing, graph construction via Joern-generated CPGs, embedding extraction, model training using Devign, and finally, explanation generation with Illuminati and integration of a visualization module for interpretability.

only reveals the underlying decision logic *given a specific model, and does not mitigate spuriousness or improve model performance.* Instead, our work unifies spuriousness mitigation and explanation in a single framework VISION to identify the truly crucial parts of the source code. While we incorporate Illuminati as an internal explainer, we complement it with an interactive, human-in-the-loop visualization module, enabling users to inspect attributions on the same input in a user-friendly manner, essential for real-world applicability.

## 3 Proposed Framework: VISION

This section presents the VISION framework [1], designed to enhance the robustness and interpretability of vulnerability detection systems. Focusing on the CWE-20 vulnerability from the PrimeVul (Ding et al. 2024), our framework generates counterfactuals—minimally edited examples with opposite labels—by leveraging LLMs to balance the training data and improve generalization. Code samples are parsed into Code Property Graphs (CPGs) using Joern, and then embedded for input to the Devign model. Interpretability is achieved through the Illuminati explainer, with an additional interactive module supporting qualitative analysis of the model's input attributions. Figure 1 illustrates the full architecture.

[1]The implementation of the VISION framework is publicly available at https://github.com/David-Egea/VISION.

```
validGlxScreen(ClientPtr client, int screen,
               __GLXscreen **pGlxScreen, int *err) {
    if (screen >= screenInfo.numScreens) {
        client->errorValue = screen;
        *err = BadValue;
        return FALSE;
    }
    *pGlxScreen = glxGetScreen(screenInfo.screens[screen]);
    return TRUE;
}
```

Figure 2: CWE-20 Improper Input Validation example. The function `validGlxScreen` checks whether the input `screen` is greater than or equal to the number of available screens, but fails to validate negative values. This omission allows invalid array access and illustrates a classic case of insufficient input validation.

### Dataset Selection: CWE-20 Vulnerability

To evaluate our framework VISION under realistic conditions, we derive our training and evaluation data from PrimeVul (Ding et al. 2024), a recently released high-quality vulnerability dataset for code language models. This dataset was curated with human-verified labels and rigorous deduplication, resulting in lower label noise and high diversity of real-world code patterns.

We focus exclusively on examples of **CWE-20 *(Improper Input Validation)*** (see (MITRE 2025)). This vulnerability class, characterized by the failure to properly sanitize or validate external inputs, was chosen for three main reasons:

1. **Clarity of semantics**. Improper input validation vulnerabilities often exhibit clear and recognizable coding patterns, such as missing boundary checks or unchecked buffer lengths, which facilitate both automated augmentation and human analysis.
2. **Data availability**. PrimeVul contains a sufficient number of CWE-20 instances (∼14k) to support both model training and reliable statistical evaluation without resorting to over-sampling or synthetic over-generation.
3. **Real-world relevance**. CWE-20 vulnerabilities remain among the most commonly exploited in practice, reinforcing the need for effective detection to strengthen software security (MITRE 2022).

A simple, illustrative example from the CWE-20 dataset is shown in Figure 2. The function `validGlxScreen` takes a `screen` index and returns a pointer to its data. However, it only checks whether `screen` is too large, neglecting negative values (or other invalid ranges), and thus allowing invalid dereferences.

**Generalization to Other Vulnerabilities and Languages.** Although this study focuses on C source code examples of CWE-20 for illustrative purposes, VISION is a general-purpose, dataset-agnostic framework. Applying it to other CWEs (e.g. CWE-416 Use-After-Free or CWE-787 Out-of-Bounds Write), or even to a mixture of vulnerabilities, would essentially follow the same process: (1) assemble a corpus of code examples labeled with the target vulnerability where any initial class imbalance is acceptable;

(2) leverage an LLM to generate counterfactuals that equalize the dataset; (3) translate all snippets into CPGs using an appropriate parser (Joern currently provides front-ends for twelve widely used languages); and (4) discard any invalid generated graphs. The downstream graph-learning and attribution components can operate without modification.

## Counterfactual Generation and Augmentation

**Definition of Counterfactuals.** First, we define a *counterfactual* as a minimally modified version of a source code function whose vulnerability label is flipped relative to the original. These modifications preserve syntactic and semantic validity while altering the vulnerability, transforming a benign sample into a vulnerable one, or vice versa. This concept draws inspiration from counterfactual explanations for tabular classification (Wachter, Mittelstadt, and Russell 2018; Verma, Dickerson, and Hines 2020; Hamman et al. 2023; Dutta et al. 2022) where counterfactual explanations are the closest point on the other side of the decision boundary. It also has intuitive connections with contrastive and counterfactual learning, where the goal is to expose the model to near-boundary examples (Kaushik, Hovy, and Lipton 2020; Ross, Marasović, and Peters 2021; Dissanayake and Dutta 2024; Temraz and Keane 2021).

Formally, given a fixed predictive model $f(\cdot)$, input graph $G = (V, E)$ with node set $V = \{v_1, v_2, \ldots, v_N\}$ and edge set $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ , original prediction $f(G) = y$ and counterfactual prediction $y_c \neq y$, a counterfactual mapping $e : G \to G'$ such that $f(e(G)) = y_c$.

We observe that counterfactuals are particularly useful in the context of vulnerability detection, because: (i) they balance the dataset by providing an equal number of examples with opposite label; and (ii) they help the model learn to distinguish subtle input validation changes that mark the presence (or absence) of a vulnerability (see Figure 3), leading to an improved understanding of the truly crucial aspects of the input. Unlike traditional augmentation methods that inject noise or generate synthetic samples from scratch, counterfactuals maintain high fidelity to real-world code, improving both training stability and downstream robustness.

**Counterfactual Generation Strategy.** We generate the counterfactuals using a prompt-based rewriting approach with the GPT-4o-mini model via the OpenAI API. Each source function was minimally modified to flip its vulnerability label, either introducing or removing a CWE-20 vulnerability, based on a prompt dynamically built from the function's original source code, label and CWE type.

Figure 3 shows an example of a counterfactual transformation. The original benign function (top) calls `net_cmd` with a fixed, safe parameter (`NULL`), ensuring no user input is involved. In the counterfactual version (bottom), a new parameter `user_input` is introduced and directly passed to `net_cmd` without validation. This edit changes the function's vulnerability status, injecting a CWE-20 Improper Input Validation flaw. Such pairs expose the model to near-identical source code that differ only in vulnerability semantics, encouraging it to learn discriminative patterns.

| Dataset Stage | Benign | Vulnerable | Total |
|---|---|---|---|
| PrimeVul | 218,529 | 6,004 | 224,533 |
| CWE-20 PrimeVul | 14,473 | 471 | 14,944 |
| **CWE-20 CFA** | **13,778** | **13,778** | **27,556** |
| – *Original* | 13,349 | 429 | 13,778 |
| – *Counterfactual* | 429 | 13,349 | 13,778 |

Table 1: CWE-20 Dataset Filtering and Balancing Summary

**Dataset Balancing and Processing Methodology.** To prevent the model from biasing towards the original or majority-class samples, we explicitly balance the dataset during the augmentation process. First, we extracted only **CWE-20 (Improper Input Validation)** examples from the full PrimeVul dataset, filtering it from 224,533 total functions down to a focused subset of 14,944 CWE-20 samples. This subset included 14,473 benign and only 471 vulnerable examples, revealing a substantial class imbalance.

To address this imbalance, we applied counterfactual generation to both benign and vulnerable samples. This strategy ensured that we were not over-representing either original or synthetic instances, effectively doubling the dataset while producing a perfectly balanced distribution of classes. Problematic or incorrectly generated examples as well as original samples whose counterfactuals could not be reliably created or validated were removed, resulting in the final dataset: CWE-20-CFA, consisting of 27,556 samples (13,778 original functions and 13,778 counterfactuals), equally divided between the two classes (Table 1).

Code samples were converted into graph representations to enable training within a GNN framework. This transformation was performed using Joern, a state-of-the-art static analysis tool that constructs Code Property Graphs (CPGs) by unifying multiple structural views of the code (Yamaguchi et al. 2014). Each function was parsed into a CPG and serialized into a structured graph format. A Word2Vec-style embedding encoder was then applied to map code tokens and edge types into continuous feature vectors.

## Base Model for Vulnerability Detection

The VISION framework builds upon the **Devign** architecture, a Graph Neural Network (GNN)-based architecture specifically designed for vulnerability detection in software source code (Zhou et al. 2019). Devign operates on Code Property Graphs (CPGs) (Yamaguchi et al. 2014), placing emphasis on the Abstract Syntax Tree (AST) to model the syntactic structure of source code. While ASTs can identify simple issues like insecure arguments, their combination with control flow graphs (CFG) and data flow graphs (DFG) enables the model to cover a wider range of vulnerabilities and learn patterns effectively.

Devign architecture includes three primary components:

1. **Graph Embedding Layer.** Converts source code into a composite graph structure, capturing multiple semantic dimensions.

Original benign code: No CWE-20 issue.

```
static int net_get_rate(struct wif *wi)
{
    struct priv_net *pn = wi_priv(wi);

    return net_cmd(pn, NET_GET_RATE, NULL, 0);
}
```

Vulnerable counterfactual example: Unvalidated `user_input` introducing a CWE-20 flaw.

```
static int net_get_rate(struct wif *wi, int user_input)
{
    struct priv_net *pn = wi_priv(wi);

    // Introduced vulnerability: accepting user input without validation
    return net_cmd(pn, NET_GET_RATE, &user_input, sizeof(user_input));
}
```

Figure 3: Illustration of a counterfactual code pair used in data augmentation. The top function is benign, safely invoking `net_cmd` with no external input. The bottom version introduces a CWE-20 (Improper Input Validation) vulnerability by replacing a fixed argument with user-provided input (`user_input`) that is passed without validation.

2. **Gated Graph Recurrent Layers (GGRU).** Extract meaningful features by iteratively propagating and aggregating information through graph nodes.
3. **Convolutional (Conv) Module.** Performs graph-level classification by effectively summarizing node embeddings into predictive vulnerability labels.

The original Devign model was extensively evaluated on manually labeled datasets from large-scale, open-source C projects such as the Linux Kernel, QEMU, Wireshark, and FFmpeg. The evaluation demonstrated Devign's significant improvement over state-of-the-art methods, achieving an average accuracy improvement of 10.51% and F1-score improvement of 8.68% (Zhou et al. 2019).

VISION uses Devign as the baseline model structure due to its proven ability to encode intricate semantic structures and effectively classify vulnerabilities at a granular level.

### Visualization Module

We developed a visualization module to support interpretability and human-in-the-loop analysis, with a particular focus on inspecting individual source code examples and explaining the behavior of the trained model through input attributions. Its primary goal is to provide evidence for the central hypothesis of this work: that counterfactual-based augmentation leads to improved model accuracy, robustness, and a better understanding of source code through more semantically meaningful attributions.

This module is built on top of the Illuminati explainer (He, Ji, and Huang 2023) and displays model predictions, confidence scores, and node-level importance values. Illuminati leverages Devign as the underlying representation for generating explanations, revealing the model's decision-making process through minimal and sufficient subgraph extraction. Predictions are color-coded—green when the model predicts correctly, and red when it misclassifies—providing immediate visual feedback on performance.

One of the key features of our visualization module is the highlighted source code view, where the original function text is color-coded based on the importance scores derived from the model explanation. A continuous color scale maps higher importance to warmer colors, allowing users to quickly identify which statements the model considered most influential. In parallel, the graph view represents the function as a map of nodes and edges that follow the program's logical flow, applying the same color scale for interpretability consistency.

Beyond direct attribution visualization, the system also offers subgraph-based interpretability. This includes positive subgraphs, where nodes are incrementally added in order of importance until the model recovers the correct prediction; negative subgraphs, where nodes are progressively removed to identify the minimal structural weaknesses that lead to prediction failure; and optimal subgraphs, which represent the subset of nodes that yield the highest overall confidence in the prediction.

An example of this process is shown in Figure 4, which compares the visualization output for an original benign function (top) and its corresponding vulnerable counterfactual (bottom). The highlighted tokens and graph regions clearly demonstrate how the model's attention shifts in response to the augmented structure.

While the visualization module is not the core contribution, it is a powerful tool for qualitative inspection, debugging, and explanation verification. It has proven particularly useful in validating whether the model's input attributions align with semantically relevant regions of the code, especially for conducting ablation studies with and without using our proposed counterfactual-based augmentation.

## 4 Experimental Evaluation

To evaluate the effectiveness of our counterfactual augmentation framework, we analyze model performance across a series of training benchmarks with varying proportions of original and counterfactual source code examples. *Evaluation is conducted using both conventional metrics, such as*

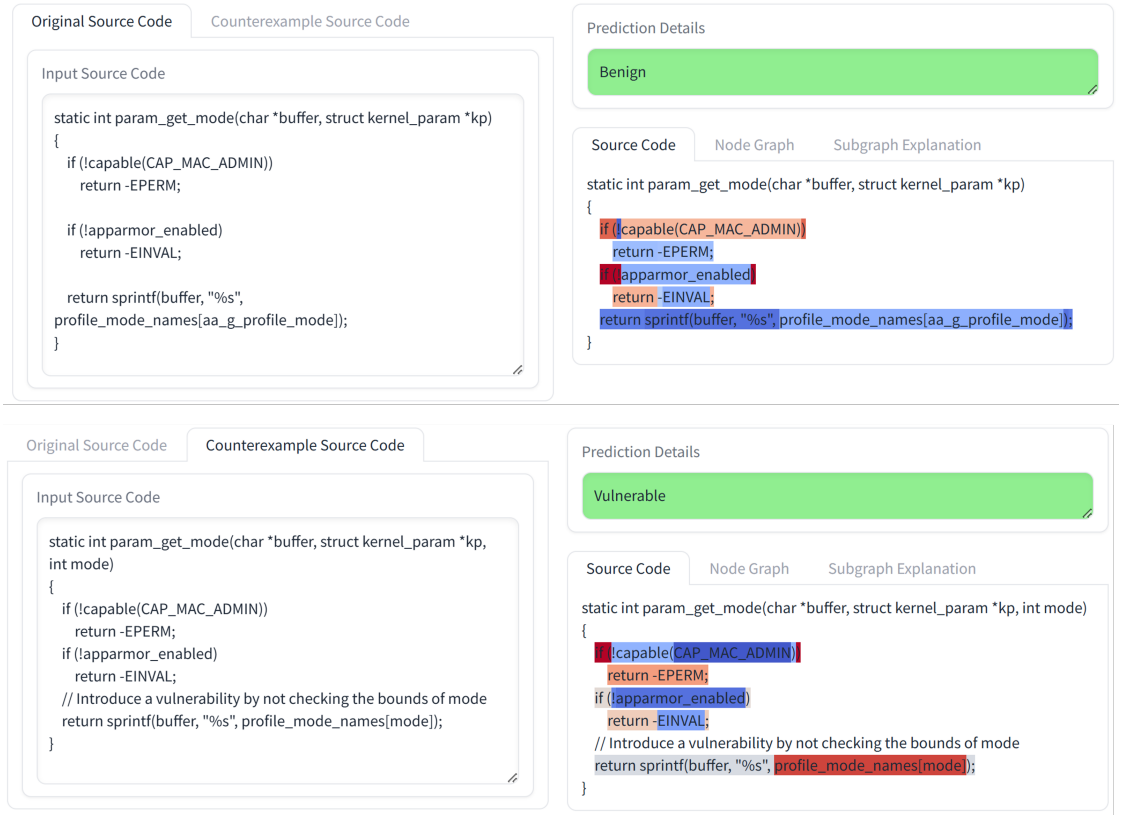


Figure 4: Integrated visualization module showcasing model predictions and explanation scores for an original benign function (top) and its vulnerable counterfactual (bottom). The right-hand side highlights tokens with attribution scores, where red intensity indicates higher importance. The module enables intuitive inspection of the model's input attribution shifts, demonstrating how the counterfactual structure influences both the prediction and explanation.

*accuracy, precision, recall, and F1-score, as well as a set of metrics specifically aimed at evaluating robustness and spuriousness mitigation. These metrics include pair-wise accuracy, worst-group accuracy, causal effect detection, embedding space neighborhood analysis, intra-class attribution variance, inter-class attribution distance, and a new node score dependency metric introduced in this work.*

## Experimental Setup

**Dataset Splitting and Augmentation Strategy.** To evaluate the impact of counterfactual-based augmentation, we construct benchmark datasets with varying ratios of original to counterfactual functions. All benchmarks are derived from the CWE-20-CFA dataset generated using the VISION framework. Each dataset contains the same total number of examples, balanced between benign and vulnerable classes, but differs in the original-to-counterfactual composition.

A fixed, balanced test set is used across all benchmarks to ensure consistent evaluation. The full dataset is first partitioned by unique IDs using an 80/10/10 train-validation-test split. Each ID is assigned exclusively to either the training or test set to preserve the integrity of original–counterfactual pairs. The test set contains both versions of each function, ensuring perfect class balance and mirrored pairings.

Training benchmarks range from 100% original to 100% counterfactual, in 10% increments. To maintain label balance within each split, examples are independently upsampled per class and data source as needed. This setup allows the model to learn from subtle syntactic and semantic differences, improving its ability to distinguish between benign and vulnerable code. Figure 5 summarizes the dataset composition. The test dataset is kept constant for all evaluations to ensure fair comparison and consists of an equal number of benign and vulnerable functions, with each original sample paired with its corresponding counterfactual.

## Performance Across Benchmarks

We report standard evaluation metrics—accuracy, precision, recall, and F1-score—for models trained under varying original/counterfactual data ratios, ensuring fair comparison through a consistent test set. As shown in Table 2, incorporating counterfactual examples consistently improves model performance across all evaluation metrics. The 100/0 configuration yields perfect precision but suffers from extremely low recall and F1-score, highlighting severe overfitting to superficial patterns. In contrast, performance peaks around the 50/50 split, where the model demonstrates strong generalization with balanced precision and recall. However,

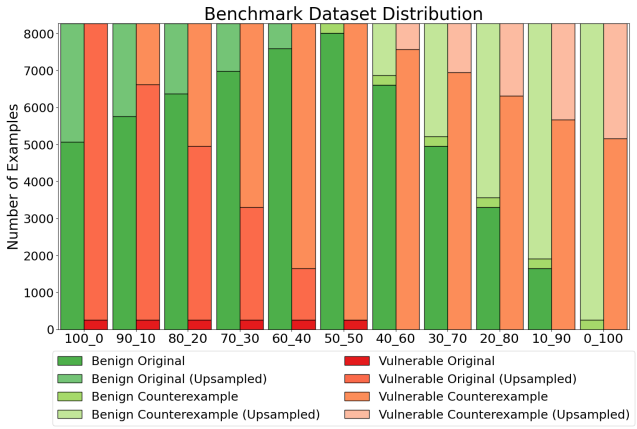


Figure 5: Benchmark dataset distribution by class and data source (original vs. counterfactual). Each bar stack shows the number of examples for benign and vulnerable samples, and the split between original, counterfactual, and upsampled sources.

| Split | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| 100/0 | 0.518 | 1.000 | 0.036 | 0.069 |
| 90/10 | 0.867 | 0.996 | 0.737 | 0.847 |
| 80/20 | 0.955 | 0.960 | 0.948 | 0.954 |
| 70/30 | 0.970 | 0.960 | 0.980 | 0.970 |
| 60/40 | **0.978** | 0.961 | **0.997** | **0.979** |
| 50/50 | 0.960 | 0.957 | 0.962 | 0.960 |
| 40/60 | 0.970 | **0.998** | 0.941 | 0.969 |
| 30/70 | 0.951 | 0.949 | 0.953 | 0.951 |
| 20/80 | 0.930 | 0.904 | 0.962 | 0.932 |
| 10/90 | 0.919 | 0.875 | 0.978 | 0.924 |
| 0/100 | 0.799 | 0.726 | 0.957 | 0.826 |

Table 2: Performance across original/counterfactual training splits on the balanced test set.

relying exclusively on synthetic data, as in the 0/100 case, leads to notable performance degradation—indicating that synthetic examples alone lack sufficient information for effective learning. Overall, these results confirm that balanced integration of counterfactuals enhances learning robustness while maintaining predictive stability.

## Robustness and Spurious Correlation Analysis

In the context of source code, spurious correlations often emerge when a model learns to associate vulnerability labels with superficial patterns that do not reflect the true semantics or security posture of the code. These misleading correlations often stem from recurring harmless syntax patterns, rather than true indicators of security flaws.

Figure 6 illustrates this phenomenon with two semantically related functions. The upper function is benign and retrieves a configuration value (`aa_g_profile_mode`) from a secure internal source. The lower function is vulnerable, as it accepts a user-controlled input (`mode`) that is directly

```c
static int param_get_mode(char *buffer, struct kernel_param *kp)
{
    if (!capable(CAP_MAC_ADMIN))
        return -EPERM;
    if (!apparmor_enabled)
        return -EINVAL;
    int mode = aa_g_profile_mode; // Potentially spurious statement
    return sprintf(buffer, "\%s", profile\_mode\_names[mode]);
}
```

```c
static int param_get_mode(char *buffer, struct kernel_param *kp,
                          int mode)
{
    if (!capable(CAP_MAC_ADMIN))
        return -EPERM;
    if (!apparmor_enabled)
        return -EINVAL;

    // Introduce a vulnerability by not checking bounds of mode
    return sprintf(buffer, "%s", profile_mode_names[mode]);
}
```

Figure 6: Illustration of spurious correlation in source code. The upper (benign) function assigns a safe internal value to `mode`, while the lower (vulnerable) version takes `mode` as unchecked user input. Without sufficient counterfactuals, a model may incorrectly associate the presence of the `mode` variable with safe behavior, failing to recognize its misuse in the vulnerable case.

passed to `sprintf()` without bounds checking. A model trained only on examples like the benign variant may erroneously learn that the presence of the mode variable is indicative of safe behavior—due to its association with sanitized sources—thereby failing to flag the vulnerable pattern when mode originates from external input.

By training on both the original and counterfactual versions of such examples, the model is exposed to minimal but security-critical changes in the code structure. This encourages learning that is grounded in true semantic distinctions, rather than dataset-specific shortcuts, and is key to mitigating spurious correlations in source code analysis.

**Pair-Wise Accuracy.** Pair-wise accuracy, introduced in (Ding et al. 2024), measures how well a model distinguishes between pairs of semantically similar functions with opposite vulnerability labels, typically an original and its minimally modified counterfactual.

Formally, a high pair-wise classification accuracy indicates that the model distinguishes between subtle code-level changes that cause label flips, rather than relying on spurious features. This metric is an indicator of the model's sensitivity to subtle, meaningful changes in source code.

Pair-wise accuracy includes four components: P-C (Pair-Correct) measures correct contrast between original and counterfactual examples; P-V (Pair-Vulnerable) and P-B (Pair-Benign) represent incorrect predictions where both functions are classified as vulnerable or benign, respectively; and P-R (Pair-Reversed) captures flipped predictions. High P-C and low P-V, P-B, and P-R indicate the model effectively distinguishes subtle, vulnerability-inducing changes without relying on spurious patterns.

As shown in Table 3, the 50/50 benchmark achieves the highest correct contrast (P-C = 95.79%), indicating that a balanced mix of original and counterfactual data helps the model focus on subtle but important vulnerability indicators.

| Split | P-C | P-V | P-B | P-R | WGA2 | WGA3 | WGA4 | WGA5 | WGA6 | WGA7 | Purity | Intra-B | Intra-V | Inter-D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 100/0 | 4.50 | 0.00 | 95.43 | 0.07 | 0.0171 | 0.0096 | 0.0073 | 0.0067 | 0.0058 | 0.0067 | 0.707 | 0.01103 | 0.01027 | 0.00061 |
| 90/10 | 74.09 | 1.38 | 23.88 | 0.65 | 0.7309 | 0.7156 | 0.7052 | 0.7126 | 0.5909 | 0.5952 | 0.907 | 0.01120 | 0.01035 | 0.00073 |
| 80/20 | 91.07 | 5.44 | 3.27 | 0.22 | **0.9115** | **0.8828** | **0.8757** | 0.8512 | 0.8444 | 0.8205 | 0.953 | 0.01096 | 0.01046 | 0.00027 |
| 70/30 | 94.63 | 4.86 | 0.36 | 0.15 | 0.9056 | 0.8745 | 0.8757 | **0.8595** | 0.8444 | 0.8205 | 0.962 | 0.01109 | 0.00995 | 0.00010 |
| 60/40 | 93.69 | 6.31 | 0.00 | **0.00** | 0.9056 | 0.8745 | 0.8703 | 0.8512 | **0.8444** | 0.8205 | **0.967** | 0.01134 | 0.01030 | 0.00010 |
| **50/50** | **95.79** | **0.44** | **0.00** | 3.77 | 0.8991 | 0.8667 | 0.8555 | 0.8087 | 0.7955 | 0.8095 | 0.944 | **0.01061** | **0.01030** | **0.00160** |
| 40/60 | 94.12 | 1.02 | 4.50 | 0.36 | 0.8471 | 0.8089 | 0.8092 | 0.7739 | 0.7955 | 0.8067 | 0.966 | 0.01122 | 0.01036 | 0.00017 |
| 30/70 | 87.52 | 8.13 | 3.85 | 0.51 | 0.8777 | 0.8400 | 0.8266 | 0.7739 | 0.7727 | 0.7857 | 0.941 | 0.01101 | 0.01010 | 0.00038 |
| 20/80 | 70.97 | 27.72 | 1.02 | 0.29 | 0.8820 | 0.8622 | 0.8497 | 0.8174 | 0.8182 | **0.8333** | 0.929 | 0.01144 | 0.01036 | 0.00028 |
| 10/90 | 77.94 | 20.54 | 0.65 | 0.87 | 0.8584 | 0.8350 | 0.8152 | 0.8265 | 0.8149 | 0.8099 | 0.910 | 0.01103 | 0.01046 | 0.00008 |
| 0/100 | 41.51 | 57.40 | 0.65 | 0.44 | 0.5398 | 0.4983 | 0.5030 | 0.4966 | 0.4754 | 0.4742 | 0.856 | 0.01122 | 0.01007 | 0.00099 |

Table 3: Comprehensive evaluation across training splits, covering robustness, generalization, and explanation quality. Metrics include: pair-wise agreement—P-C (correct contrast), P-V (both predicted vulnerable), P-B (both predicted benign), and P-R (flipped predictions); higher P-C and lower P-V/P-B/P-R indicate better discrimination. Worst-Group Accuracy (WGA, $k$=2–7): higher is better, reflects subgroup robustness. Neighborhood Purity: higher values indicate stronger class consistency in the embedding space and better semantic separation. Attribution metrics include Intra-class Attribution Variance (lower is better, measures consistency) and Inter-class Attribution Distance (higher is better, reflects class separability).

Extremes like 100/0 or 0/100 result in higher confusion and misclassification, suggesting that spurious correlations dominate when training is biased toward only one data type.

**Worst-Group Accuracy.** Worst-Group Accuracy (WGA) is a robustness metric that captures a model's weakest performance across latent subgroups defined by both structural code patterns and class labels (Idrissi et al. 2022). It reflects how well the model avoids overfitting to spurious correlations and maintains reliable predictions even for hard-to-classify regions of the data.

Since no explicit spurious attributes are available, we adopt an unsupervised approach to define these subgroups. First, we extract latent code embeddings from the trained model and apply K-means clustering to identify groups of structurally or stylistically similar functions. Each function is then assigned to a subgroup based on its cluster ID and ground truth label. Groups with fewer than 1% of the total data are discarded to avoid instability. The WGA is computed as the lowest classification accuracy among the remaining subgroups.

Given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ and a trained model $f$, we define latent groups via unsupervised clustering. Let $g_i \in \mathcal{G}$ denote the group assignment of sample $x_i$, determined by the intersection of its K-means cluster ID and true label $y_i$.

For each group $g \in \mathcal{G}$ with size $|g| > 0.01N$, we compute its accuracy:

$$\text{Acc}(g) = \frac{1}{|g|} \sum_{i \in g} [f(x_i) = y_i]$$

The Worst-Group Accuracy (WGA) is then defined as:

$$\text{WGA} = \min_{g \in \mathcal{G},\ |g| > 0.01N} \text{Acc}(g)$$

Table 3 presents the WGA values across different original/counterfactual training splits and clustering granularities. The 100/0 model performs poorly across all k-values, with WGA below 2%, indicating brittle generalization and overfitting to spurious patterns. Models trained with moderate counterfactual integration (e.g., 80/20 to 60/40) achieve the highest and most stable WGA values, consistently above 85%. Performance slightly drops for fully synthetic or highly imbalanced configurations (0/100 or 50/50). Because $k = 2$ almost perfectly separates the two ground-truth classes, $\text{WGA}_2$ is invariably the largest and tracks the class-level Purity metric closely (Pearson $r \approx 0.88$ across splits). For $k > 2$ each class is subdivided into rarer stylistic clusters, so the worst-group accuracy naturally falls. Splits with the highest $\text{WGA}_2$ also show the lowest intra-class attribution variance. *These results support the hypothesis that counterfactual augmentation mitigates spurious correlations by improving the model's consistency across diverse code structures.*

**Neighborhood Analysis in Embedding Space.** Neighborhood purity measures the extent to which embeddings of samples cluster with others of the same class in latent space. Higher purity indicates stronger semantic consistency and suggests that the model has learned to organize representations based on meaningful vulnerability patterns rather than spurious shortcuts.

We compute the Neighborhood Purity Score using a k-nearest neighbor (kNN) strategy over graph-level embeddings, and report the average class consistency across all samples.

The calculated scores for all training splits appear in Table 3 to show how class consistency in the embedding space changes with different levels of counterfactual augmentation

Neighborhood purity increases sharply as counterfactuals are introduced, peaking around the 60/40 split. The 100/0 model shows the lowest purity (0.71), suggesting understructured embeddings due to overfitting. While purity remains high for most augmented settings, it slightly declines for heavily synthetic configurations (e.g., 0/100), indicating reduced semantic clustering. These results suggest that mod-

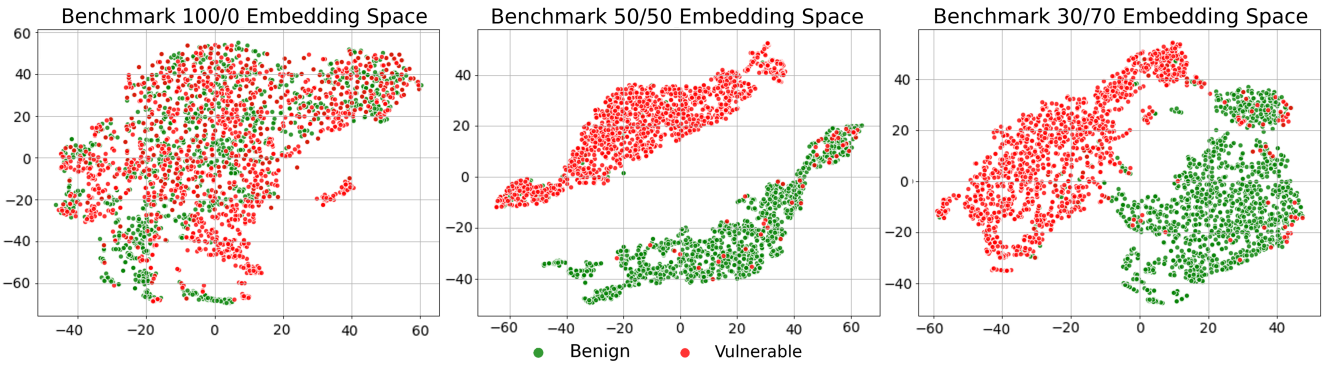


Figure 7: t-SNE projections of graph embeddings for benchmarks 100/0 (left), 50/50 (center), and 30/70 (right). Green points represent benign samples, and red points represent vulnerable samples. These visualizations illustrate how different original/-counterfactual splits influence the spatial distribution and separation of vulnerability classes.

erate augmentation improves embedding structure without inducing shortcut learning.

To further analyze how embedding space evolves under different augmentation ratios, we visualize graph-level representations using t-SNE projections for selected benchmarks (Figure 7). The 0/100 model shows heavily entangled clusters, with benign and vulnerable examples largely mixed—indicating poor separation and potential reliance on non-semantic features. In contrast, the 50/50 benchmark yields a much cleaner separation between classes, suggesting that balanced counterfactual training enables the model to form more meaningful latent representations. The 30/70 exhibits intermediate behavior: vulnerable and benign examples are not as distinctly partitioned as in the 50/50 configuration. This visualization qualitatively supports the neighborhood purity findings and highlights how counterfactual augmentation enhances the model's ability to semantically organize code representations in latent space.

**Intra-Class Attribution Variance and Inter-Class Attribution Distance.** In this work, we also propose two other attribution-based metrics to evaluate the consistency and discriminability of model explanations.

The metrics are as follows:

- **Intra-Class Attribution Variance** measures how stable the explainer attributions are across different samples of the same class. High variance suggests that the model may be relying on sample-specific or potentially spurious patterns, while low variance indicates more consistent reasoning.
- **Inter-Class Attribution Distance** quantifies the difference between the average attribution vectors of benign and vulnerable samples. A larger distance suggests better separability between the explanation patterns of both classes.

These metrics are computed using attribution vectors generated by the Illuminati explainer. They reflect not only how a model performs, but also how it reasons—providing a more nuanced evaluation of explanation quality, model behavior, and robustness to dataset bias.

As shown in Table 3, intra-class variance remains relatively stable across benchmarks, but the 50/50 configuration achieves the highest inter-class attribution distance—indicating the clearest semantic separation in model reasoning. This supports that balanced augmentation not only boosts predictive accuracy but also improves explanation quality by helping the model focus on truly semantically relevant and discriminative features.

**Node Score Dependency.** Lastly, we introduce Node Score Dependency, a novel metric to analyze how the importance of each node in a graph depends on others in the context of GNN-based explanation. This metric captures inter-node attribution dynamics by quantifying how attribution scores change when a specific node is removed.

For each node $i$, we temporarily mask or remove it from the graph and recompute the attribution scores using the post hoc explainer. The difference in the importance of any other node $j \neq i$ reflects the degree to which node $j$'s contribution relies on node $i$. This yields a node dependency matrix $M \in \mathrm{R}^{n \times n}$, where:

$$M_{i,j} = \left| \text{score}_j^{\text{orig}} - \text{score}_j^{(i\text{-removed})} \right|$$

Here, $M_{i,j}$ represents the absolute change in the importance score of node $j$ when node $i$ is removed. Diagonal entries $M_{i,i}$ capture the self-dependence of each node, while large off-diagonal values reflect strong cross-node influence.

This metric offers several practical use cases. It enhances interpretability by identifying globally or locally dominant nodes within prediction explanations and detecting spurious correlations—such as when semantically irrelevant or constant nodes unduly influence outcomes. It supports model debugging by uncovering shortcut behaviors or fragile reasoning patterns and informs robustness analysis by guiding graph perturbations to assess model stability and sensitivity.

Next, we apply this proposed analysis across different benchmarks on the same vulnerable function used in Figure 6, allowing visual comparison of attribution dynamics under varied training regimes.

The heatmap for the 100/0 benchmark reveals a narrow

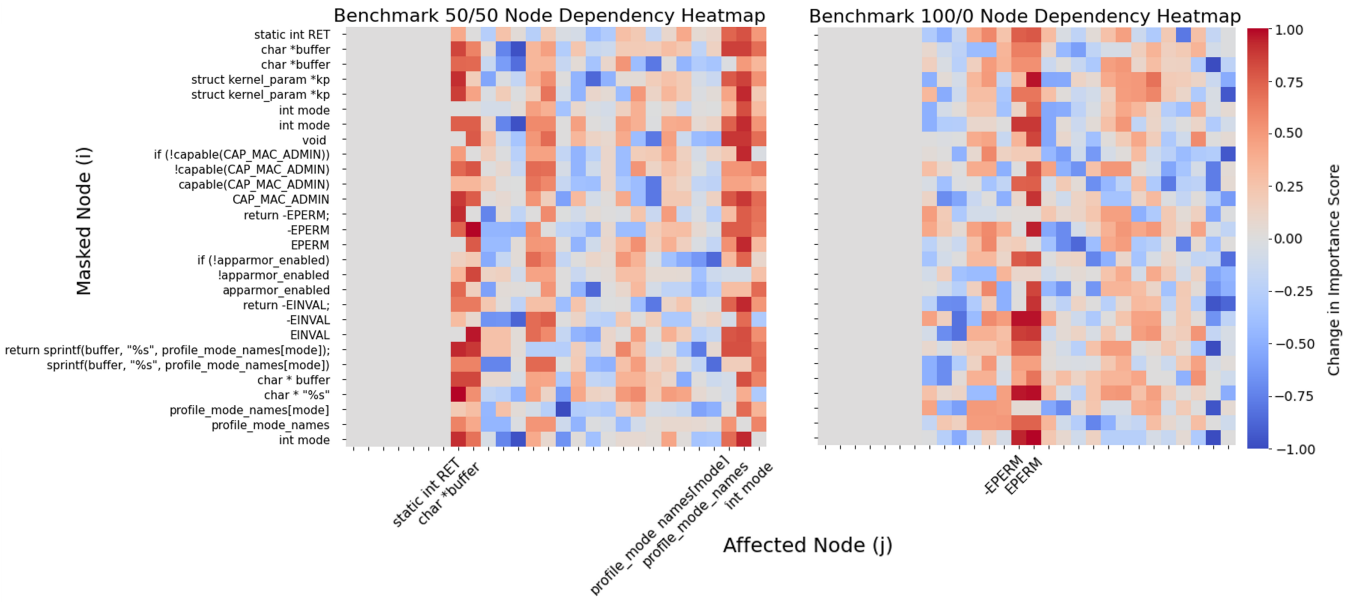


Figure 8: Node Score Dependency heatmaps for the same vulnerable function under two training regimes: Left: Benchmark 100/0 and Right: Benchmark 50/50. Each heatmap shows how masking one node (rows) affects the attribution scores of others (columns). Red indicates increased influence; blue indicates reduced importance. The 100/0 model exhibits high focus on spurious context nodes, while 50/50 shows more meaningful attribution alignment.

attribution dependency pattern, with notably high sensitivity centered around the variable `EPERM`. This suggests that the model disproportionately relies on this node when forming its prediction. However, this behavior likely reflects a spurious correlation, as `EPERM`—a standard error code—does not fundamentally determine the presence of a CWE-20 vulnerability. Conversely, critical vulnerability-relevant components such as `mode`, `profile_mode_names`, and `profile_mode_names[mode]`, which directly relate to the improper input validation flaw, show little to no influence on the attribution of other nodes. This indicates that the model fails to capture the semantic importance of these core vulnerability-inducing elements.

In contrast, the 50/50 benchmark, trained on a balanced mix of original and counterfactual examples, demonstrates a more structured and semantically aligned dependency map. Here, masking `profile_mode_names` or `profile_mode_names[mode]` significantly alters the attribution of related nodes, suggesting that the model has internalized the interdependence between these components. Furthermore, conditional statements and their associated return values also exhibit logical attribution interactions, implying better recognition of control-flow implications.

*These observations underscore the effectiveness of the proposed node dependency metric in revealing both spurious and semantically meaningful attribution patterns.* The comparison between benchmarks demonstrates that counterfactual data augmentation improves the model's learning consistency, encouraging reliance on true vulnerability signals rather than superficial correlations. Across all metrics, this collectively highlights counterfactual augmentation as a robust strategy for mitigating spurious learning and enhancing the stability, generalization, and interpretability of GNN-based vulnerability detection systems.

## 5 Conclusions and Future Work

The research presents a single unified framework, VISION, which detects vulnerabilities in source code through counterfactual data augmentation while providing interpretability. The method generates paired examples through systematic, minimal semantic changes to prevent models from learning spurious correlations that appear in noisy or imbalanced datasets. The model learns to detect actual vulnerability patterns rather than relying on superficial code features through GNN training on counterfactual pairs. The framework uses graph-based explainability to reveal important decision-making components while providing an interactive visualization module for human-in-the-loop analysis.

We also point out two limitations. First, VISION has been evaluated exclusively on CWE-20 in this study. However, extending VISION to new CWEs entails the same process: gather labeled examples, generate counterfactuals with an LLM, and translate the source code into CPGs. The downstream modules require no further adjustment. Second, the use of LLM-generated counterfactuals may occasionally introduce unrealistic or noisy modifications. Future work will therefore (i) evaluate the framework across a broader set of CWEs and programming languages to assess generalization; and (ii) integrate semantic-preserving generation approaches together with formal verification to assess the correctness of all generated counterfactuals.

## Acknowledgments

D. Egea was a research intern at the University of Maryland, supported through a partnership between the Office of Global Engineering Leadership (OGEL) and Universidad Pontificia Comillas. B. Halder and S. Dutta were supported in part by Google Gift Funding and NSF CAREER Award No. 2340006. The authors also thank Yanjun Fu, Faisal Hamman, and Pasan Dissanayake for their valuable feedback and suggestions.

## References

Bell, S. J.; and Wang, S. 2024. The Multiple Dimensions of Spuriousness in Machine Learning. arXiv:2411.04696.

Chen, Y.; Bian, Y.; Zhou, K.; Xie, B.; Han, B.; and Cheng, J. 2023a. Does invariant graph learning via environment augmentation learn invariance? *Advances in Neural Information Processing Systems*, 36: 71486–71519.

Chen, Y.; Ding, Z.; Alowain, L.; Chen, X.; and Wagner, D. 2023b. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. arXiv:2304.00409.

Chernis, B.; and Verma, R. 2018. Machine Learning Methods for Software Vulnerability Detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, IWSPA '18, 31–39. New York, NY, USA: Association for Computing Machinery. ISBN 9781450356343.

Chu, Z.; Wan, Y.; Li, Q.; Wu, Y.; Zhang, H.; Sui, Y.; Xu, G.; and Jin, H. 2024. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '24, 389–401. ACM.

Croft, R.; Babar, M. A.; and Kholoosi, M. 2023. Data Quality for Software Vulnerability Datasets. arXiv:2301.05456.

Daneshvar, S. S.; Nong, Y.; Yang, X.; Wang, S.; and Cai, H. 2024. Exploring RAG-based Vulnerability Augmentation with LLMs. arXiv:2408.04125.

Ding, K.; Xu, Z.; Tong, H.; and Liu, H. 2022. Data augmentation for deep graph learning: A survey. *ACM SIGKDD Explorations Newsletter*, 24(2): 61–77.

Ding, Y.; Fu, Y.; Ibrahim, O.; Sitawarin, C.; Chen, X.; Alomair, B.; Wagner, D.; Ray, B.; and Chen, Y. 2024. Vulnerability Detection with Code Language Models: How Far Are We? arXiv:2403.18624.

Dissanayake, P.; and Dutta, S. 2024. Model reconstruction using counterfactual explanations: A perspective from polytope theory. *Advances in Neural Information Processing Systems*, 37: 83397–83429.

Dutta, S.; Long, J.; Mishra, S.; Tilli, C.; and Magazzeni, D. 2022. Robust counterfactual explanations for tree-based ensembles. In *International conference on machine learning*, 5742–5756. PMLR.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155.

Ganz, T.; Imgrund, E.; Harterich, M.; and Rieck, K. 2023. CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery. In *Data and Applications Security and Privacy XXXVII: 37th Annual IFIP WG 11.3 Conference, DBSec 2023, Sophia-Antipolis, France, July 19–21, 2023, Proceedings*, 282–301. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-031-37585-9.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; Tufano, M.; Deng, S. K.; Clement, C.; Drain, D.; Sundaresan, N.; Yin, J.; Jiang, D.; and Zhou, M. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366.

Guo, Y.; and Bettaieb, S. 2023. An Investigation of Quality Issues in Vulnerability Detection Datasets. In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 29–33. IEEE.

Haig, B. D. 2003. What is a spurious correlation? *Understanding Statistics: Statistical Issues in Psychology, Education, and the Social Sciences*, 2(2): 125–132.

Halder, B.; Hamman, F.; Dissanayake, P.; Zhang, Q.; Sucholutsky, I.; and Dutta, S. 2024. Quantifying spuriousness of biased datasets using partial information decomposition. *arXiv preprint arXiv:2407.00482*.

Hamman, F.; Noorani, E.; Mishra, S.; Magazzeni, D.; and Dutta, S. 2023. Robust Counterfactual Explanations for Neural Networks With Probabilistic Guarantees. In *International Conference on Machine Learning*, 12351–12367. PMLR.

He, H.; Ji, Y.; and Huang, H. H. 2023. Illuminati: Towards Explaining Graph Neural Networks for Cybersecurity Analysis. arXiv:2303.14836.

Idrissi, B. Y.; Arjovsky, M.; Pezeshki, M.; and Lopez-Paz, D. 2022. Simple data balancing achieves competitive worst-group-accuracy. arXiv:2110.14503.

Kaushik, D.; Hovy, E.; and Lipton, Z. C. 2020. Learning the Difference that Makes a Difference with Counterfactually-Augmented Data. arXiv:1909.12434.

Kong, K.; Li, G.; Ding, M.; Wu, Z.; Zhu, C.; Ghanem, B.; Taylor, G.; and Goldstein, T. 2022. Robust optimization as data augmentation for large-scale graphs. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 60–69.

Li, Y.; Wang, S.; and Nguyen, T. N. 2021. 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*, 292–303. ACM.

Liu, G.; Zhao, T.; Xu, J.; Luo, T.; and Jiang, M. 2022a. Graph rationalization with environment-based augmentations. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 1069–1078.

Liu, L.; Li, Z.; Wen, Y.; and Chen, P. 2022b. Investigating the Impact of Vulnerability Datasets on Deep Learning-Based Vulnerability Detectors. *PeerJ Computer Science*, 8: e975.

Liu, S.; Ma, W.; Wang, J.; Xie, X.; Feng, R.; and Liu, Y. 2024. Enhancing Code Vulnerability Detection via Vulnerability-Preserving Data Augmentation. arXiv:2404.09599.

Lundberg, S.; and Lee, S.-I. 2017. A Unified Approach to Interpreting Model Predictions. arXiv:1705.07874.

MITRE. 2022. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

MITRE. 2025. CWE-20: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html. Accessed: 2025-05-23.

Neuhaus, S.; Zimmermann, T.; Holler, C.; and Zeller, A. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, 529–540. New York, NY, USA: Association for Computing Machinery. ISBN 9781595937032.

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. arXiv:1602.04938.

Ross, A.; Marasović, A.; and Peters, M. E. 2021. Explaining NLP Models via Minimal Contrastive Editing (MiCE). arXiv:2012.13985.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1): 61–80.

Sharma, D. K.; Mishra, J.; Singh, A.; Govil, R.; Srivastava, G.; and Lin, J. C.-W. 2022. Explainable Artificial Intelligence for Cybersecurity. *Computers and Electrical Engineering*, 103: 108356.

Steinmann, D.; Divo, F.; Kraus, M.; Wüst, A.; Struppek, L.; Friedrich, F.; and Kersting, K. 2024. Navigating Shortcuts, Spurious Correlations, and Confounders: From Origins via Detection to Mitigation. arXiv:2412.05152.

Sultana, S.; Afreen, S.; and Eisty, N. U. 2024. Code Vulnerability Detection: A Comparative Analysis of Emerging Large Language Models. arXiv:2409.10490.

Temraz, M.; and Keane, M. T. 2021. Solving the Class Imbalance Problem Using a Counterfactual Method for Data Augmentation. arXiv:2111.03516.

Verma, S.; Dickerson, J.; and Hines, K. 2020. Counterfactual explanations for machine learning: A review. *arXiv preprint arXiv:2010.10596*.

Vu, M. N.; and Thai, M. T. 2020. PGM-Explainer: Probabilistic Graphical Model Explanations for Graph Neural Networks. arXiv:2010.05788.

Wachter, S.; Mittelstadt, B.; and Russell, C. 2018. Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR. arXiv:1711.00399.

Wang, J.; Huang, M.; Nie, Y.; Kuang, X.; Li, X.; and Zhong, W. 2023. Fine-Grained Source Code Vulnerability Detection via Graph Neural Networks.

Wu, Y.; Zou, D.; Dou, S.; Yang, W.; Xu, D.; and Jin, H. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2365–2376.

Yamaguchi, F.; Golde, N.; Arp, D.; and Rieck, K. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*, 590–604.

Ye, W.; Zheng, G.; Cao, X.; Ma, Y.; and Zhang, A. 2024. Spurious Correlations in Machine Learning: A Survey. arXiv:2402.12715.

Ying, R.; Bourgeois, D.; You, J.; Zitnik, M.; and Leskovec, J. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *NeurIPS*.

Zhou, X.; Zhang, T.; and Lo, D. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. arXiv:2401.15468.

Zhou, Y.; Liu, S.; Siow, J.; Du, X.; and Liu, Y. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. arXiv:1909.03496.

Ziems, N.; and Wu, S. 2021. Security Vulnerability Detection Using Deep Learning Natural Language Processing. arXiv:2105.02388.