COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MASTER'S DEGREE IN TELECOMMUNICATIONS ENGINEERING

MASTER THESIS

IMPROVING ROBUSTNESS IN VULNERABILITY DETECTION VIA COUNTERFACTUAL AUGMENTATION

Author:
David Egea Hernández

Supervisor:
Sanghamitra Dutta

Madrid
June 15, 2025

I declare, under my own responsibility, that the Project submitted with the title

Improving Robustness in Vulnerability Detection via Counterfactual Augmentation

carried out at the ICAI School of Engineering – Universidad Pontificia Comillas during the academic year 2024/2025 my own work, original and unpublished, and has not been submitted previously for any other purpose.

The Project is not, either wholly or partially, a plagiarism of another work, and all information taken from other documents is properly referenced.

Signed: David Egea Hernández Date: 15/06/2025

Authorization for the submission of the project

PROJECT SUPERVISOR

Signed: Sanghamitra Dutta Date: 15/06/2025

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MASTER'S DEGREE IN TELECOMMUNICATIONS ENGINEERING

MASTER THESIS

IMPROVING ROBUSTNESS IN VULNERABILITY DETECTION VIA COUNTERFACTUAL AUGMENTATION

Author:
David Egea Hernández

Supervisor:
Sanghamitra Dutta

Madrid
June 15, 2025

Acknowledgements

My sincere gratitude to Dr. Sanghamitra Dutta for all her guidance and support throughout this research. It has been a true privilege to collaborate with her, making this an experience in which I learned more than I had ever imagined.

I would also like to thank Barproda for her continued help, whose valuable insights and advice were fundamental in shaping the direction of this work.

This research would not have been possible without the collaboration between Universidad Pontificia Comillas (ICAI) and the University of Maryland (UMD). I am thankful to both institutions for enabling this international research experience and for their commitment to supporting students through initiatives like this one.

I feel very fortunate to have made all the good friends I did during this time; it was their companionship that made this experience truly special and unforgettable.

Lastly, I am especially grateful to my family, and in particular my parents, for their invaluable support throughout this overseas experience, which marks the final stage of a challenging but incredibly rewarding and beautiful educational journey.

MEJORANDO LA ROBUSTEZ EN LA DETECCIÓN DE VULNERABILIDADES MEDIANTE AUMENTACIÓN CONTRAFACTUAL

Autor: Egea Hernández, David.

Director: Dutta, Sanghamitra.

Entidad Colaboradora: University of Maryland.

Resumen

La detección automatizada de vulnerabilidades en el código fuente es una tarea crítica de ciberseguridad que afecta a la seguridad y fiabilidad del software moderno. Las herramientas tradicionales de análisis estático y dinámico a menudo no logran capturar estructuras sintácticas complejas y lógicas semánticas sutiles. Los avances recientes en aprendizaje automático, especialmente las Redes Neuronales Gráficas (GNNs), muestran potencial para aprender relaciones estructurales y semánticas en el código. Sin embargo, su rendimiento se ve limitado por el desequilibrio de datos, el ruido en las etiquetas y las correlaciones espurias—patrones que asocian erróneamente características superficiales con etiquetas de vulnerabilidad—lo que socava su capacidad de generalización ante fallos del mundo real. Para abordar esto, este trabajo presenta VISION (*Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION*), un marco unificado para mejorar la robustez y la interpretabilidad en la detección de vulnerabilidades. VISION aprovecha Modelos de Lenguaje de Gran Escala (LLMs) para generar contraejemplos—funciones mínimamente editadas con etiquetas de vulnerabilidad invertidas—proporcionando al modelo contrastes significativos entre muestras reales y sintéticas. La GNN Devign se emplea como arquitectura del modelo base, combinada con un *explainer* para atribuciones detalladas en el grafo y un módulo de visualización para análisis humano-en-el-bucle. Evaluado sobre CWE-20 (*Improper Input Validation*), VISION logra mejoras sustanciales en precisión, generalización y calidad de las explicaciones, con avances notables en precisión por pares y en el peor subgrupo. Además, se proponen nuevas métricas de interpretabilidad y se publica el benchmark CWE-20-CFA, con más de 27.000 ejemplos balanceados. En conjunto, VISION ofrece un enfoque novedoso y eficaz para mitigar el aprendizaje espurio y avanzar hacia una IA transparente y confiable para el desarrollo de software seguro.

Introducción

Las vulnerabilidades de software se encuentran entre las debilidades más explotadas en ciberseguridad, sirviendo a menudo como el principal punto de entrada

para los atacantes. Garantizar su detección temprana y precisa es vital para proteger la integridad y funcionalidad de las infraestructuras digitales [1]. Los enfoques tradicionales, como el análisis estático y dinámico del código, tienen dificultades para abordar la complejidad sintáctica y semántica del software moderno, lo que limita su escalabilidad y capacidad de generalización [2]. Como respuesta, las Redes Neuronales Gráficas (GNNs) han surgido como una alternativa prometedora. Al modelar el código fuente como grafos—capturando elementos como árboles sintácticos, flujos de control y dependencias de datos—las GNNs pueden aprender estructuras significativas del programa y respaldar la detección basada en datos [3], [4]. Sistemas como Devign [5] han demostrado que las GNNs son capaces de aprender características semánticas complejas y superar a los métodos tradicionales en tareas de predicción de vulnerabilidades.

A pesar de su potencial, los modelos basados en GNN se ven considerablemente limitados por problemas en los datos de entrenamiento. Los conjuntos de datos de referencia suelen presentar muestras duplicadas, etiquetas ruidosas o inconsistentes y un fuerte desequilibrio entre clases [6], [7], [8]. Estos problemas con frecuencia provocan que los modelos aprendan correlaciones espurias—asociaciones estadísticas engañosas que no reflejan la semántica de seguridad subyacente [9], [10], [11]. Como consecuencia, estos modelos pueden parecer efectivos durante la evaluación, pero fracasan al generalizar sobre código real no visto. Además, sus procesos de decisión suelen ser opacos, lo que dificulta entender qué aspectos de la entrada influyen en sus predicciones. Sin una interpretabilidad clara, la fiabilidad y la confianza en estos sistemas se ven limitadas, especialmente en contextos de seguridad críticos.

Para abordar estos desafíos, este trabajo presenta **VISION** (*Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION*), un marco que busca mejorar tanto la robustez como la interpretabilidad en la detección de vulnerabilidades. Este enfoque utiliza contraejemplos de código—funciones que han sido mínimamente editadas para invertir su etiqueta de vulnerabilidad, preservando la corrección sintáctica y semántica. Por ejemplo, transformar una llamada de `strcpy(dest, "fixed_string")` a `strcpy(dest, user_input)` introduce una vulnerabilidad al permitir una entrada no validada, a pesar de que ambas líneas sean estructuralmente similares. Estos ejemplos se generaron mediante Modelos de Lenguaje de Gran Tamaño (LLMs), siguiendo una estrategia de reescritura basada en prompts. El objetivo principal es exponer a la GNN a variaciones semánticas sutiles, ayudando al modelo a centrarse en patrones de seguridad significativos en lugar de artefactos específicos del conjunto de datos.

El modelo utilizado en VISION se basa en la arquitectura Devign [5], entrenada con muestras emparejadas originales y contraejemplos para mejorar la discriminación entre código seguro y vulnerable. Para aumentar la transparencia, el marco integra el *explainer* Illuminati [12], que proporciona atribuciones basadas en grafos que resaltan los elementos del código más influyentes. Esto se complementa con un módulo de visualización que facilita el análisis humano.

VISION aborda una carencia clave en interpretabilidad para la detección de vulnerabilidades. Mientras que herramientas como GNNExplainer [13]

y CFExplainer [14] ofrecen atribuciones de características y localización de vulnerabilidades, no mitigan el aprendizaje espurio. En cambio, VISION unifica aumentación de datos e interpretabilidad, mejorando el razonamiento del modelo y ofreciendo a los profesionales una interfaz intuitiva para explorar y validar resultados—apoyando así un flujo de trabajo de IA más transparente y confiable para el desarrollo de software seguro [15].

La evaluación experimental se centra en la categoría CWE-20 de la Common Weakness Enumeration—*Improper Input Validation*—que se encuentra entre los tipos de vulnerabilidades más frecuentes y peligrosos [16]. Los resultados muestran una mejora significativa en el rendimiento predictivo y la capacidad de generalización, con un aumento en la precisión de contraste por pares del 4.5 % al 95.8 % y en la precisión del peor subgrupo del 0.7 % a más del 85 %, lo que indica una reducción sustancial del aprendizaje espurio.

Definición y Objetivos

Este proyecto se sitúa en la intersección entre la seguridad del software y la inteligencia artificial explicable, con el objetivo de abordar limitaciones críticas en los actuales sistemas de detección de vulnerabilidades basados en aprendizaje automático. Estos sistemas, especialmente aquellos basados en Redes Neuronales Gráficas (GNNs), suelen sufrir de sobreajuste a correlaciones espurias provocadas por ruido en los datos, desequilibrio de clases y similitudes estructurales entre código benigno y vulnerable. Además, la falta de interpretabilidad en estos modelos dificulta su adopción en entornos reales de ciberseguridad, donde la confianza y la transparencia son esenciales.

El marco propuesto, VISION (*Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION*), está diseñado para mejorar tanto la robustez como la explicabilidad en la detección de vulnerabilidades en código basada en GNNs. Lo consigue mediante la generación e integración de contraejemplos—funciones de código mínimamente editadas con etiquetas de vulnerabilidad invertidas—para guiar al modelo hacia el aprendizaje de distinciones de seguridad significativas. El marco también incorpora un módulo de visualización para la interpretación del modelo, aprovechando atribuciones basadas en grafos que permiten identificar regiones del código influyentes.

Objetivos del Proyecto

- Generar un conjunto de datos aumentado (CWE-20-CFA) centrado en la vulnerabilidad CWE-20 mediante la implementación de una canalización de generación de contraejemplos basada en prompts utilizando LLMs.

- Aplicar una metodología de evaluación comparativa entrenando GNNs con diferentes proporciones de ejemplos originales/contraejemplos (por ejemplo, 90/10, 50/50) y evaluándolos mediante métricas de precisión, contraste por pares y robustez para analizar el impacto de la aumentación contrafactual.

- Implementar un módulo de explicación basado en grafos que resalte componentes influyentes del código utilizando un *explainer* post hoc con puntuaciones de importancia a nivel de nodo, permitiendo la atribución visual y facilitando la comprensión del modelo en entornos humano-en-el-bucle.
- Demostrar que el uso combinado de datos contrafactuales y técnicas de explicación reduce el aprendizaje espurio, mejora la generalización y refuerza la confianza en los sistemas de detección de vulnerabilidades.

Descripción del Marco de Trabajo

Esta sección presenta el marco VISION, que mejora la robustez y la interpretabilidad en la detección de vulnerabilidades. Enfocado en CWE-20, utiliza contraejemplos generados por LLMs para equilibrar los datos de entrenamiento y mejorar la capacidad de generalización. Las muestras de código se convierten en *Code Property Graphs* (CPGs) utilizando Joern y se vectorizan para el modelo Devign. Las explicaciones se generan utilizando el *explainer* Illuminati, con un módulo interactivo que permite un análisis cualitativo de atribuciones. La Figura S1 muestra la arquitectura completa.

Selección del Conjunto de Datos: Vulnerabilidad CWE-20

Para evaluar el marco VISION en condiciones realistas, este trabajo utiliza el conjunto de datos PrimeVul [6], un benchmark recientemente publicado y rigurosamente curado para la detección de vulnerabilidades en código fuente. PrimeVul ofrece etiquetas verificadas por humanos y estrictos estándares de eliminación de duplicados, proporcionando una base fiable para el entrenamiento y evaluación con menor ruido en las etiquetas y alta diversidad semántica.

Este proyecto se centra específicamente en la clase de vulnerabilidad CWE-20, conocida como Validación Incorrecta de Entradas (*Improper Input Validation*) [16], seleccionada según los siguientes criterios:

1. Claridad semántica. Estas vulnerabilidades suelen seguir patrones bien definidos—como la ausencia de comprobaciones de límites o buffers sin validar—lo que las hace adecuadas tanto para el análisis automatizado como manual.
2. Cantidad de datos suficiente. El conjunto de datos PrimeVul incluye aproximadamente 14.000 instancias de CWE-20, lo que permite un entrenamiento y evaluación robustos del modelo sin recurrir a sobre-muestreo o datos sintéticos en exceso.
3. Impacto práctico. CWE-20 sigue siendo uno de los tipos de vulnerabilidad más explotados, lo que resalta la importancia de su detección precisa en contextos reales de seguridad del software [17].

Un ejemplo que ilustra CWE-20 (Figura S2) es la función `validGlxScreen`, que valida si un índice de pantalla se encuentra dentro de los límites válidos. Aunque verifica los límites superiores, no rechaza los valores negativos, lo que puede

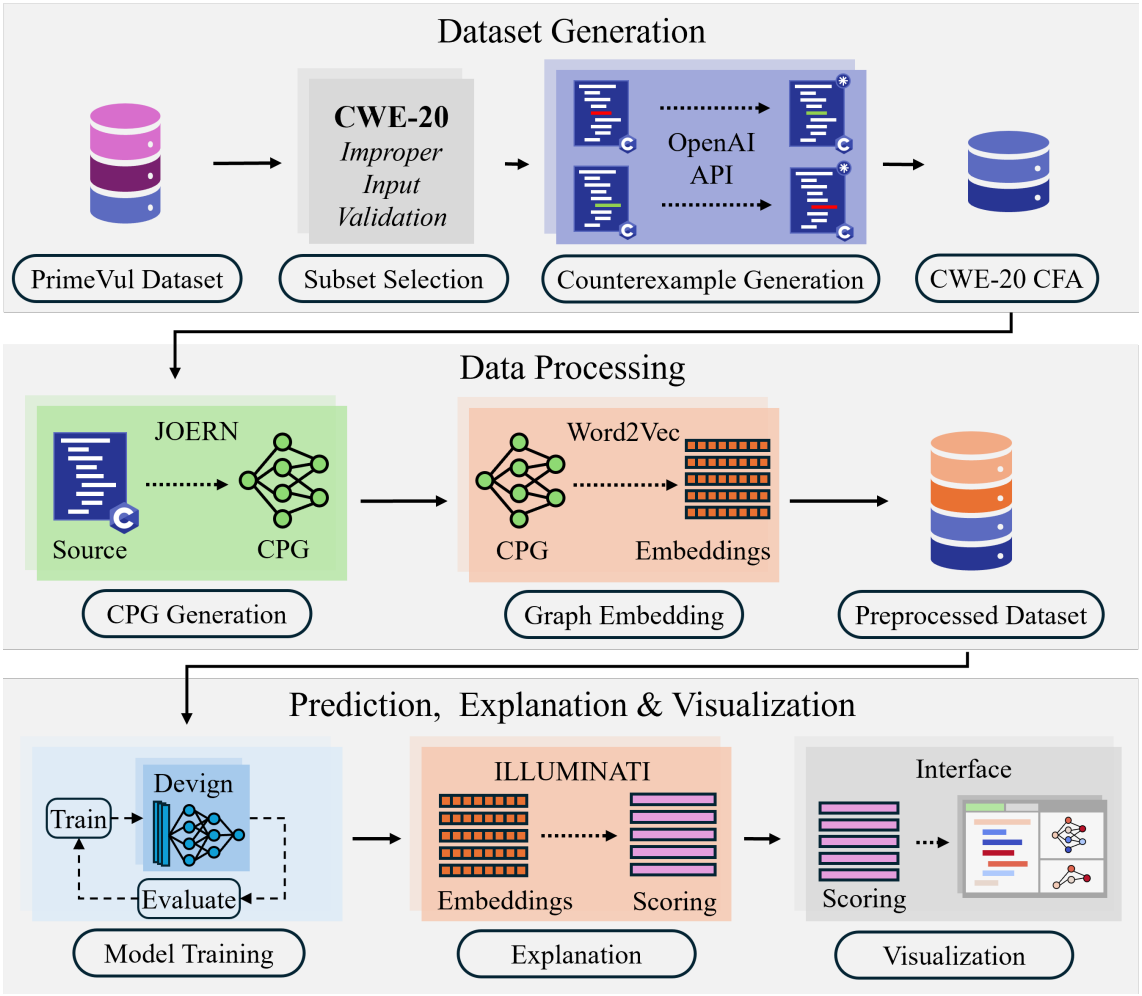


Figura S1: Resumen de la arquitectura del marco VISION. A partir del conjunto de datos fuente original, la canalización de extremo a extremo incluye el filtrado por CWE-20, la generación de contraejemplos para el balanceo de clases, la construcción de CPGs, la extracción de embeddings, el entrenamiento del modelo y la generación de explicaciones, todo ello apoyado por el módulo de visualización para una interpretación comprensible.

provocar accesos fuera de rango. Este tipo de descuido ejemplifica la naturaleza sutil pero crítica de los fallos en la validación de entradas y motiva la necesidad de mecanismos de detección precisos.

Generación y Aumentación con Contraejemplos

Un contraejemplo se define como una versión mínimamente modificada de una función de código cuya etiqueta de vulnerabilidad se invierte con respecto a la original. Estas modificaciones mantienen la corrección sintáctica y semántica, pero alteran la condición de vulnerabilidad—transformando una muestra benigna en una vulnerable o viceversa. Este enfoque se inspira en el razonamiento contrafactual dentro de la inteligencia artificial explicable [18] y se basa en ideas del aprendizaje contrastivo que exponen al modelo a ejemplos en la frontera de decisión [19], [20].

```
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;
}
```

Figura S2: Ejemplo de *Improper Input Validation* CWE-20. La función `validGlxScreen` verifica que el índice `screen` no exceda el número de pantallas disponibles, pero omite comprobar si el valor es negativo. Esta omisión puede provocar accesos inválidos a arrays y ejemplifica una vulnerabilidad típica de validación incorrecta de entradas.

En el contexto de este trabajo, los contraejemplos cumplen dos propósitos fundamentales: equilibran el conjunto de datos entre clases y guían al modelo para que reconozca diferencias semánticas sutiles indicativas de vulnerabilidades reales. A diferencia de los métodos tradicionales de aumentación, que pueden introducir ruido o patrones artificiales, los contraejemplos mantienen una alta fidelidad con las estructuras de código originales, favoreciendo un entrenamiento estable y una mejor generalización.

Los contraejemplos se generaron utilizando una estrategia de reescritura basada en prompts mediante el modelo OpenAI GPT-4o-mini, utilizando prompts construidos dinámicamente que incluyen el fragmento de código original, su etiqueta y contexto del tipo CWE. Cada función fue modificada para invertir su etiqueta de vulnerabilidad, introduciendo o eliminando una falla de tipo CWE-20. Un ejemplo se muestra en la Figura S3, donde la inserción de una entrada de usuario no validada en una función benigna da lugar a una vulnerabilidad realista.

Código original benigno: Sin vulnerabilidad CWE-20.

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

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

Código de contraejemplo vulnerable a CWE-20: Entrada `user_input` sin validar.

```
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));
}
```

Figura S3: Ejemplo de un par de funciones contraejemplares. La función superior es benigna y realiza una llamada segura a `net_cmd` con un argumento fijo. La versión inferior introduce una vulnerabilidad CWE-20 (*Improper Input Validation*) al sustituir el argumento fijo por una entrada de usuario no validada (`user_input`), demostrando un cambio semántico mínimo que altera la etiqueta de seguridad.

Para construir un conjunto de entrenamiento balanceado, se extrajeron 14.944 muestras CWE-20 del conjunto PrimeVul, que presentaba un fuerte desequilibrio a favor de funciones benignas. A partir de ambas clases se generaron contraejemplos, eliminando aquellas muestras problemáticas o inválidas. El conjunto final—CWE-20-CFA—contiene 27.556 funciones: 13.778 originales y 13.778 contraejemplos, distribuidos equitativamente entre clases benignas y vulnerables (ver Tabla S1).

Cuadro S1: Resumen del Filtrado y Balanceo del Conjunto de Datos CWE-20

Conjunto de Datos	**Benignos**	**Vulnerables**	**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

Modelo Base para la Detección de Vulnerabilidades

El marco VISION implementa Devign [5] como la arquitectura base de Red Neuronal de Grafos para la clasificación de vulnerabilidades. Devign está específicamente diseñado para el análisis de código fuente y opera sobre Grafos de Propiedades del Código (CPGs) [2], los cuales integran árboles de sintaxis abstracta (AST), grafos de flujo de control (CFG) y de flujo de datos (DFG) en una representación unificada.

La arquitectura consta de tres componentes principales: (1) una capa de incrustación de grafos que codifica características de nodos y aristas en representaciones latentes; (2) Unidades Recurrentes de Grafos con Puertas (GGRU) que propagan y agregan información a través de la estructura del grafo; y (3) un módulo convolucional que realiza la clasificación a nivel de grafo resumiendo las incrustaciones de nodos aprendidas.

El modelo Devign fue originalmente evaluado en proyectos C de gran escala y etiquetado manualmente (por ejemplo, Linux Kernel, QEMU, Wireshark, FFmpeg), mostrando mejoras notables en el rendimiento respecto a métodos anteriores. VISION adopta Devign como línea base debido a su efectividad demostrada al capturar patrones semánticos detallados para la clasificación de vulnerabilidades.

Módulo de Visualización

Para apoyar la interpretabilidad y el análisis humano, este proyecto integra un módulo de visualización dedicado, diseñado para inspeccionar ejemplos individuales de código fuente y explicar el comportamiento del modelo entrenado mediante atribuciones de entrada. El módulo sirve para validar la hipótesis central de que la aumentación basada en contraejemplos mejora no solo la precisión y la robustez, sino también la alineación semántica de las explicaciones del modelo.

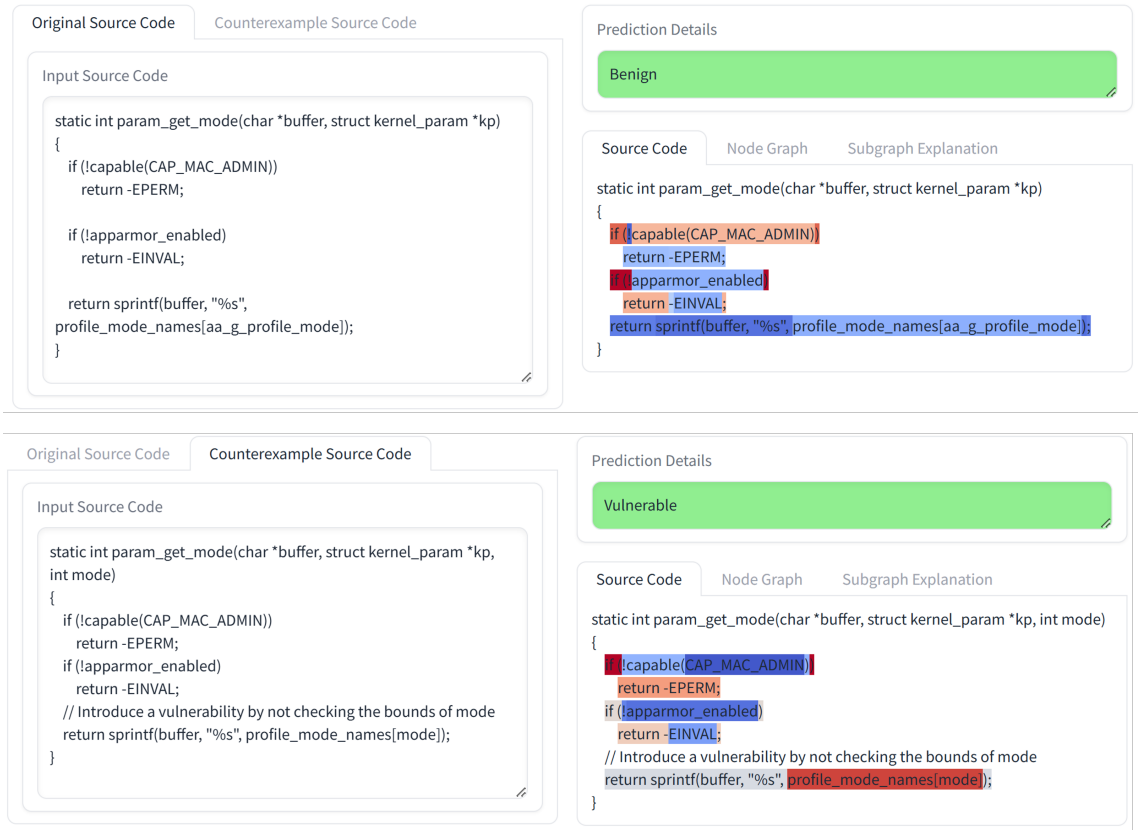


Figura S4: Módulo de visualización integrado que muestra las predicciones del modelo y las puntuaciones de explicación para una función benigna original (arriba) y su contraejemplo vulnerable (abajo). Las puntuaciones de atribución se visualizan a la derecha, con sombreado rojo que indica la importancia de cada token. El módulo facilita una exploración intuitiva de los cambios en atribuciones, revelando cómo la modificación contrafactual afecta tanto a la predicción como al razonamiento del modelo.

El sistema se construye sobre el explicador Illuminati [12], que genera puntuaciones de importancia a nivel de nodo extrayendo subgrafos mínimos y suficientes desde el modelo GNN. Las predicciones se codifican visualmente: las clasificaciones correctas se muestran en verde, mientras que las incorrectas aparecen en rojo, proporcionando retroalimentación inmediata sobre el rendimiento del modelo para el análisis humano.

Se ofrecen dos vistas sincronizadas para cada función: una visualización del código fuente resaltado y un mapa de nodos basado en grafo. En ambas vistas se emplea una escala de colores continua para indicar la importancia relativa de los tokens o nodos, ayudando al usuario a identificar qué elementos fueron más influyentes en la predicción del modelo.

Como se ilustra en la Figura S4, la visualización revela cómo varía el enfoque del modelo entre variantes benignas y vulnerables de una función, apoyando una comprensión más profunda de la lógica de decisión influida por la estructura contrafactual.

Experimentos y Resultados

Para evaluar la efectividad del marco VISION, se realizó una evaluación exhaustiva sobre una serie de benchmarks de entrenamiento compuestos por diferentes proporciones de funciones originales y contraejemplares. Todos los experimentos se basaron en el conjunto de datos CWE-20-CFA y se diseñaron para medir no solo la precisión predictiva, sino también la robustez frente a correlaciones espurias.

Se crearon múltiples configuraciones de benchmark, desde un 100 % de datos originales hasta un 100 % de datos contraejemplares, en incrementos del 10 %. Cada configuración mantuvo constante el número de muestras de entrenamiento y una distribución perfectamente balanceada entre clases (benignas vs. vulnerables). Las particiones del conjunto de datos siguieron una proporción 80/10/10 para entrenamiento, validación y prueba, conservando la integridad del emparejamiento al asignar cada ID de función exclusivamente a una única partición. El conjunto de prueba se mantuvo fijo en todas las evaluaciones para garantizar la consistencia e incluyó tanto la versión original como la contraejemplar de cada función.

Se utilizaron métricas estándar—precisión, exactitud, exhaustividad y puntuación F1—para evaluar el rendimiento general, mientras que métricas adicionales como precisión por pares, precisión del peor subgrupo, análisis del espacio de embeddings y puntuaciones basadas en atribuciones se emplearon para evaluar la robustez y la capacidad de generalización. Los resultados muestran que el rendimiento mejora de forma constante con la aumentación contrafactual, alcanzando su punto máximo en torno a una proporción 50/50. La configuración con 100 % de datos originales logró una alta precisión, pero con una baja exhaustividad, lo que indica sobreajuste a patrones superficiales del código. Por el contrario, la configuración con 0 % de datos originales (totalmente sintética) redujo notablemente el rendimiento, confirmando que los ejemplos reales son esenciales para un entrenamiento efectivo. Estas tendencias validan que una integración equilibrada de contraejemplos promueve un aprendizaje más robusto y generalizable (ver Tabla S2).

Análisis de Robustez y Correlación Espuria

La precisión por pares (*pair-wise accuracy*) evalúa la capacidad del modelo para distinguir entre funciones de código semánticamente similares con etiquetas de vulnerabilidad opuestas—normalmente una original y su contraejemplo. Una puntuación alta indica que el modelo es sensible a ediciones de código sutiles pero significativas que alteran el estado de vulnerabilidad, en lugar de basarse en patrones superficiales. La métrica se descompone en cuatro resultados: Par Correcto (P-C), Par Vulnerable (P-V), Par Benigno (P-B) y Par Invertido (P-R). Los modelos óptimos presentan valores altos en P-C y bajos en las otras tres categorías. Los resultados muestran que la proporción 50/50 entre originales y contraejemplos alcanza la mayor precisión por contraste (95.79 %), lo que confirma que una aumentación balanceada ayuda al modelo a aprender distinciones relevantes para la seguridad.

La Precisión del Peor Subgrupo (WGA, por sus siglas en inglés—*Worst Group*

Cuadro S2: Evaluación integral a través de divisiones de entrenamiento sobre el conjunto de prueba balanceado. La sección izquierda reporta precisión, exactitud, exhaustividad y puntuación F1. La sección derecha evalúa robustez, generalización y calidad de las explicaciones. Las métricas incluyen: P-C (ambas correctas), P-V (ambas predichas como vulnerables), P-B (ambas predichas como benignas) y P-R (predicciones invertidas); WGA (Precisión del Peor Subgrupo, $k=4$) refleja la robustez frente a subgrupos. La Pureza del Vecindario mide la consistencia de clases en el espacio de embeddings. La Varianza de Atribución Intra-clase (cuanto menor, mejor) y la Distancia de Atribución Inter-clase (cuanto mayor, mejor) evalúan la consistencia de las explicaciones y la separabilidad entre clases.

Split	**Acc**	**Prec**	**Rec**	**F1**	**P-C**	**P-V**	**P-B**	**P-R**	**WGA4**	**Purity**	**Intra-B**	**Intra-V**	**Inter-D**
100/0	0.518	1.000	0.036	0.069	4.50	0.00	95.43	0.07	0.0073	0.707	0.01103	0.01027	0.00061
90/10	0.867	0.996	0.737	0.847	74.09	1.38	23.88	0.65	0.7052	0.907	0.01120	0.01035	0.00073
80/20	0.955	0.960	0.948	0.954	91.07	5.44	3.27	0.22	0.8757	0.953	0.01096	0.01046	0.00027
70/30	0.970	0.960	0.980	0.970	94.63	4.86	0.36	0.15	**0.8757**	0.962	0.01109	0.00995	0.00010
60/40	**0.978**	0.961	**0.997**	**0.979**	93.69	6.31	0.00	**0.00**	0.8703	**0.967**	0.01134	0.01030	0.00010
50/50	0.960	0.957	0.962	0.960	**95.79**	**0.44**	**0.00**	3.77	0.8555	0.944	**0.01061**	**0.01030**	**0.00160**
40/60	0.970	**0.998**	0.941	0.969	94.12	1.02	4.50	0.36	0.8092	0.966	0.01122	0.01036	0.00017
30/70	0.951	0.949	0.953	0.951	87.52	8.13	3.85	0.51	0.8266	0.941	0.01101	0.01010	0.00038
20/80	0.930	0.904	0.962	0.932	70.97	27.72	1.02	0.29	0.8497	0.929	0.01144	0.01036	0.00028
10/90	0.919	0.875	0.978	0.924	77.94	20.54	0.65	0.87	0.8152	0.910	0.01103	0.01046	0.00008
0/100	0.799	0.726	0.957	0.826	41.51	57.40	0.65	0.44	0.5030	0.856	0.01122	0.01007	0.00099

Accuracy) mide el rendimiento más débil del modelo entre subgrupos descubiertos automáticamente mediante agrupamiento de embeddings latentes, basados en la estructura del código y la clase. Refleja la robustez al indicar si el modelo se sobreajusta a patrones dominantes mientras falla en regiones minoritarias o difíciles de generalizar. Los grupos se definen mediante K-means e intersecan con las etiquetas reales; la WGA se calcula como la menor precisión de clasificación entre los grupos suficientemente grandes. Los modelos entrenados con una aumentación moderada de contraejemplos presentan los valores más altos y estables de WGA (¿85 %), lo que confirma que los contraejemplos mejoran la generalización en subgrupos.

La Pureza de los vecinos (*Neighborhood Purity*) evalúa qué tan bien los embeddings de grafos aprendidos se agrupan según las etiquetas reales de clase. Se calcula mediante la consistencia de los k vecinos más cercanos (kNN) en el espacio latente y sirve como indicador de si el modelo organiza las representaciones de manera semántica o se basa en atajos espurios. La integración balanceada de contraejemplos mejora la pureza, alcanzando su puntuación más alta en la configuración 60/40, mientras que el modelo 100/0 muestra baja pureza debido a embeddings mal estructurados. Visualizaciones mediante t-SNE confirman además que los modelos con aumentación moderada generan una separación de clases más clara en el espacio latente, lo que sugiere que los contraejemplos favorecen un aprendizaje de representaciones más significativo (ver Figura S5).

Las métricas basadas en atribuciones evalúan la consistencia de las explicaciones y la separabilidad entre clases. La varianza intra-clase mide cuán similares son las atribuciones del explicador entre muestras de la misma clase—una menor varianza implica un razonamiento más consistente. La distancia inter-clase

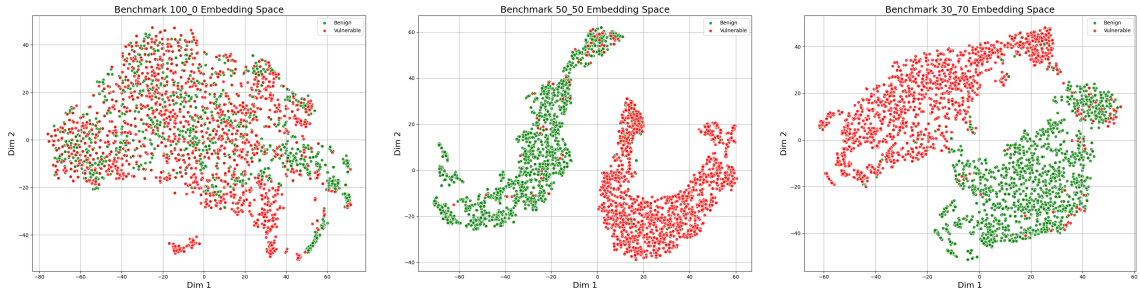


Figura S5: Proyecciones t-SNE de los embeddings de grafos para los benchmarks 100/0 (izquierda), 50/50 (centro) y 30/70 (derecha). Los puntos verdes representan muestras benignas, y los puntos rojos representan muestras vulnerables. Estas visualizaciones ilustran cómo las diferentes proporciones de originales/contraejemplos influyen en la distribución espacial y la separación de las clases de vulnerabilidad.

cuantifica cuán distintas son las atribuciones promedio entre muestras benignas y vulnerables—valores más altos sugieren fronteras conceptuales más claras. Los resultados indican que la varianza intra-clase se mantiene relativamente estable, mientras que la distancia inter-clase alcanza el máximo en el benchmark 50/50.

La Dependencia entre Puntuaciones de Nodos (*Node Score Dependency*) analiza la influencia entre nodos en las explicaciones generadas por GNN, midiendo cómo cambian las puntuaciones de importancia de un nodo al eliminar otro. Esto genera una matriz de dependencia que revela entrelazamientos en las atribuciones. En los modelos entrenados sin contraejemplos (100/0), se observa un enfoque espurio sobre tokens irrelevantes, mientras que los nodos realmente relevantes permanecen sin influencia. En contraste, el modelo 50/50 exhibe dependencias estructuradas entre componentes que inducen vulnerabilidades, reflejando un razonamiento más preciso y consciente del contexto (ver Figura S6). Esta métrica ofrece información tanto sobre la interpretabilidad como sobre la robustez del modelo al identificar trayectorias de razonamiento frágiles o basadas en atajos.

Conclusiones

Este trabajo presenta VISION, un marco para la detección de vulnerabilidades en código fuente que mejora la robustez y la interpretabilidad mediante aumentación de datos contrafactuales. Al generar modificaciones de código semánticamente válidas que invierten las etiquetas de vulnerabilidad, el marco expone a las Redes Neuronales Gráficas a patrones sutiles pero significativos, reduciendo la dependencia de correlaciones espurias. Para reforzar la interpretabilidad, VISION incorpora explicaciones basadas en grafos a través de atribuciones a nivel de nodo e incluye un módulo interactivo de visualización para facilitar el análisis humano.

La evaluación a lo largo de múltiples benchmarks demuestra que la aumentación contrafactual conduce a mejoras consistentes en precisión, robustez y calidad de las explicaciones. En particular, el marco alcanza una alta precisión por pares y en el peor subgrupo, generando espacios de embedding y patrones de atribución

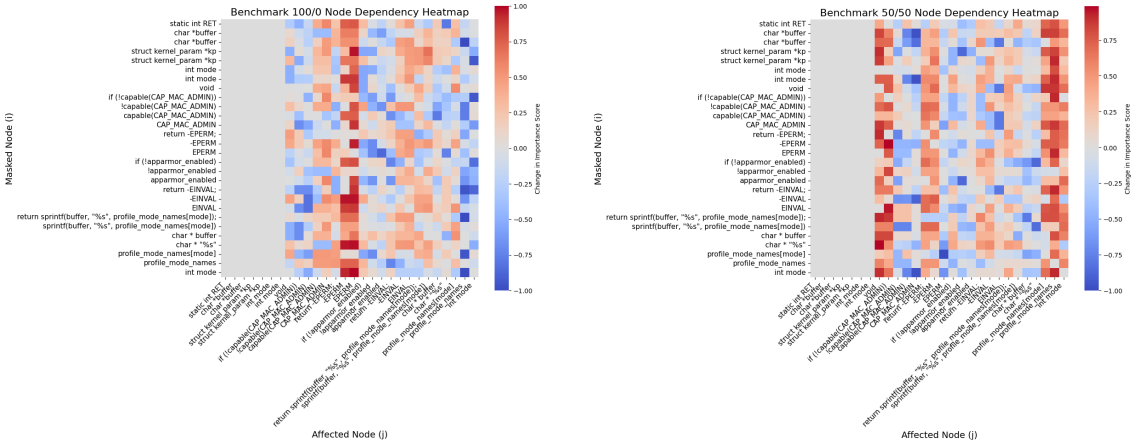


Figura S6: Mapas de calor de la Dependencia entre Puntuaciones de Nodos para la misma función vulnerable bajo dos regímenes de entrenamiento: Izquierda: Benchmark 100/0 y Derecha: Benchmark 50/50. Cada mapa muestra cómo enmascarar un nodo (filas) afecta las puntuaciones de atribución de los demás (columnas). El rojo indica un aumento en la influencia; el azul indica una disminución en la importancia. El modelo 100/0 exhibe un enfoque elevado en nodos de contexto espurio, mientras que el modelo 50/50 muestra una alineación de atribuciones más significativa.

más estructurados semánticamente. Estos resultados validan VISION como una estrategia eficaz para mitigar el aprendizaje basado en atajos.

No obstante, dos limitaciones restringen actualmente la generalidad del marco. Primero, los experimentos se limitan a una única clase de vulnerabilidad—CWE-20 (Validación Incorrecta de Entradas)—lo cual puede limitar su aplicabilidad a otros contextos de seguridad. Segundo, el uso de contraejemplos generados por modelos de lenguaje puede introducir en ocasiones ediciones poco realistas.

El trabajo futuro se centrará en extender VISION a clases adicionales de vulnerabilidad y en explorar estrategias alternativas para la generación de contraejemplos, como el uso de verificación formal. También se investigarán métodos para evaluar y reforzar la corrección semántica de los contraejemplos, con el objetivo de facilitar una adopción más amplia de este marco para la construcción de sistemas de IA robustos y transparentes en ciberseguridad.

Bibliografía

[1] Boris Chernis and Rakesh Verma. 2018. *Machine Learning Methods for Software Vulnerability Detection.* In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics (IWSPA '18)*, ACM, New York, NY, USA, pp. 31–39. DOI: 10.1145/3180445.3180453.

[2] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. *Modeling and Discovering Vulnerabilities with Code Property Graphs.* In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*, IEEE, pp. 590–604. DOI: 10.1109/SP.2014.44.

[3] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. *The Graph Neural Network Model. IEEE Transactions on Neural Networks*, 20(1): 61–80. DOI: 10.1109/TNN.2008.2005605.

[4] Jingjing Wang, Minhuan Huang, Yuanping Nie, Xiaohui Kuang, Xiang Li, and Wenjing Zhong. 2023. *Fine-Grained Source Code Vulnerability Detection via Graph Neural Networks.* Available at: https://openreview.net/forum?id=S5RYm-9Q4o.

[5] 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.* arXiv preprint arXiv:1909.03496 [cs.SE].

[6] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. *Vulnerability Detection with Code Language Models: How Far Are We?* arXiv preprint arXiv:2403.18624 [cs.SE].

[7] Yuejun Guo and Seifeddine Bettaieb. 2023. *An Investigation of Quality Issues in Vulnerability Detection Datasets.* In *Proceedings of the 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, pp. 29–33. DOI: 10.1109/EuroSPW59978.2023.00008.

[8] Roland Croft, M. Ali Babar, and Mehdi Kholoosi. 2023. *Data Quality for Software Vulnerability Datasets.* arXiv preprint arXiv:2301.05456 [cs.SE].

[9] Wenqian Ye, Guangtao Zheng, Xu Cao, Yunsheng Ma, and Aidong Zhang. 2024. *Spurious Correlations in Machine Learning: A Survey.* arXiv preprint arXiv:2402.12715 [cs.LG].

[10] Samuel J. Bell and Skyler Wang. 2024. *The Multiple Dimensions of Spuriousness in Machine Learning*. arXiv preprint arXiv:2411.04696 [cs.LG].

[11] David Steinmann, Felix Divo, Maurice Kraus, Antonia Wüst, Lukas Struppek, Felix Friedrich, and Kristian Kersting. 2024. *Navigating Shortcuts, Spurious Correlations, and Confounders: From Origins via Detection to Mitigation*. arXiv preprint arXiv:2412.05152 [cs.LG].

[12] Haoyu He, Yuede Ji, and H. Howie Huang. 2023. *Illuminati: Towards Explaining Graph Neural Networks for Cybersecurity Analysis*. arXiv preprint arXiv:2303.14836 [cs.LG].

[13] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. *GNNExplainer: Generating Explanations for Graph Neural Networks*. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*.

[14] Zhaoyang Chu, Yao Wan, Qian Li, Yang Wu, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 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)*, ACM, pp. 389–401. DOI: 10.1145/3650212.3652136.

[15] Deepak Kumar Sharma, Jahanavi Mishra, Aeshit Singh, Raghav Govil, Gautam Srivastava, and Jerry Chun-Wei Lin. 2022. *Explainable Artificial Intelligence for Cybersecurity. Computers and Electrical Engineering*, 103:108356. DOI: 10.1016/j.compeleceng.2022.108356.

[16] MITRE. 2025. *CWE-20: Improper Input Validation*. Available at: https://cwe.mitre.org/data/definitions/20.html. Accessed: 2025-06-03.

[17] MITRE. 2022. *2022 CWE Top 25 Most Dangerous Software Weaknesses*. Available at: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[18] Sandra Wachter, Brent Mittelstadt, and Chris Russell. 2018. *Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR*. arXiv preprint arXiv:1711.00399 [cs.AI].

[19] Divyansh Kaushik, Eduard Hovy, and Zachary C. Lipton. 2020. *Learning the Difference that Makes a Difference with Counterfactually-Augmented Data*. arXiv preprint arXiv:1909.12434 [cs.CL].

[20] Alexis Ross, Ana Marasović, and Matthew E. Peters. 2021. *Explaining NLP Models via Minimal Contrastive Editing (MiCE)*. arXiv preprint arXiv:2012.13985 [cs.CL].

IMPROVING ROBUSTNESS IN VULNERABILITY DETECTION VIA COUNTERFACTUAL AUGMENTATION

Author: Egea Hernández, David.

Supervisor: Dutta, Sanghamitra.

Collaborating entity: University of Maryland.

Abstract

Automated source code vulnerability detection is a critical cybersecurity task that impacts the safety and reliability of modern software. Traditional static and dynamic analysis tools often fail to capture complex syntactic structures and subtle semantic logic. Recent advances in machine learning, especially Graph Neural Networks (GNNs), show promise in learning structural and semantic code relationships. However, their performance is limited by data imbalance, label noise, and spurious correlations—patterns that wrongly associate superficial features with vulnerability labels—undermining generalization to real-world flaws. To address this, this work introduces VISION (Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION), a unified framework for improving robustness and interpretability in vulnerability detection. VISION leverages Large Language Models (LLMs) to generate counterfactual examples—minimally edited functions with flipped vulnerability labels—providing the model with meaningful contrasts between real and synthetic samples. The Devign GNN serves as the architecture for the base model, paired with an explainer for fine-grained graph attributions and a visualization module for human-in-the-loop analysis. Evaluated on CWE-20 (Improper Input Validation), VISION achieves substantial gains in accuracy, generalization, and explanation quality, with notable improvements in pair-wise and worst-group accuracy. Additionally, new interpretability metrics are proposed and the CWE-20-CFA dataset benchmark is released, featuring over 27,000 balanced examples. Overall, VISION offers a novel and effective approach to mitigating spurious learning and advancing transparent, trustworthy AI for secure software development.

Introduction

Software vulnerabilities are among the most exploited weaknesses in cybersecurity, often serving as the primary entry point for attackers. Ensuring their early and precise detection is vital to protect the integrity and functionality of digital infrastructures [1]. Traditional approaches such as static and dynamic code analysis struggle with the syntactic and semantic complexity of modern software, limiting their scalability and generalization [2]. In response, Graph Neural Networks (GNNs) have emerged as a powerful alternative. By modeling source

code as graphs—capturing elements like syntax trees, control flow, and data dependencies—GNNs can learn meaningful program structures and support data-driven detection [3], [4]. Systems such as Devign [5] have demonstrated that GNNs are capable of learning rich semantic features and outperforming traditional methods in vulnerability prediction tasks.

Despite their promise, GNN-based models are significantly hindered by issues in training data. Benchmark datasets often suffer from duplicated samples, noisy or inconsistent labels, and strong class imbalance [6], [7], [8]. These problems frequently result in models learning spurious correlations—misleading statistical associations that do not reflect the underlying security semantics [9], [10], [11]. As a result, such models may appear effective during evaluation but fail to generalize to unseen, real-world code. Moreover, their decision processes often remain opaque, making it unclear what aspects of the input influence their predictions. Without clear interpretability, the reliability and trustworthiness of these systems are limited, especially in high-stakes security contexts.

To address these challenges, this work introduces **VISION** (Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION), a framework that aims to improve both robustness and interpretability in vulnerability detection. This approach uses counterfactual code examples—functions that have been minimally edited to flip their vulnerability label while preserving syntactic and semantic correctness. For instance, transforming a call from `strcpy(dest, "fixed_string")` to `strcpy(dest, user_input)` introduces a vulnerability by allowing unvalidated input, despite the two lines being structurally similar. These examples were generated using Large Language Models (LLMs), following a prompt-based rewriting strategy. The main goal is to expose the GNN to subtle semantic variations, helping it focus on meaningful security patterns rather than dataset-specific artifacts.

The model used in VISION is based on the Devign architecture [5], which is trained on paired original and counterfactual samples to improve discrimination between secure and vulnerable code. To enhance transparency, the framework integrates the Illuminati explainer [12], which provides graph-based attributions highlighting the most influential code elements. This is further supported by a visualization module that facilitates human-in-the-loop analysis.

VISION addresses a key gap in explainability for vulnerability detection. While tools like GNNExplainer [13] and CFExplainer [14] provide feature attributions and vulnerability localization, they do not mitigate spurious learning. In contrast, VISION unifies data augmentation and interpretability, improving model reasoning and offering practitioners an intuitive interface to explore and validate outputs—supporting a more transparent and trustworthy AI pipeline for secure software development [15].

The experimental evaluation focuses on the Common Weakness Enumeration CWE-20 category—Improper Input Validation—which is among the most frequent and dangerous types of vulnerabilities [16]. Results show significantly improved

predictive performance and generalization, with pairwise contrast accuracy rising from 4.5% to 95.8% and worst-group accuracy from 0.7% to over 85%, indicating a substantial reduction in spurious learning.

Definition and Objectives

This project is situated at the intersection of software security and explainable artificial intelligence, with the goal of addressing critical limitations in current machine learning-based vulnerability detection systems. These systems, particularly those based on Graph Neural Networks (GNNs), often suffer from overfitting to spurious correlations caused by dataset noise, imbalance, and structural similarity between benign and vulnerable code. Additionally, the lack of interpretability in such models hinders their adoption in real-world cybersecurity settings, where trust and transparency are essential.

The proposed framework, VISION (Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION), is designed to enhance both the robustness and explainability of GNN-based code vulnerability detection. It does so by generating and integrating counterfactual examples—minimally edited code functions with flipped vulnerability labels—to guide the model toward learning meaningful security distinctions. The framework also incorporates a visualization module for model interpretation, leveraging graph-based attributions to identify influential code regions.

Project Objectives

- Generate an augmented dataset (CWE-20-CFA) focused on the CWE-20 vulnerability by implementing a prompt-based counterfactual generation pipeline using LLMs.
- Apply a benchmarking methodology by training GNNs on varying original/counterfactual splits (e.g., 90/10, 50/50) and evaluating them using accuracy, pairwise contrast, and robustness metrics to assess the impact of counterfactual augmentation.
- Implement a graph-based explanation module that highlights influential code components using a post hoc explainer with node-level importance scores, enabling visual attribution and supporting human-in-the-loop understanding of model predictions.
- Demonstrate that the combined use of counterfactual data and explanation techniques reduces spurious learning, improves generalization, and enhances the trustworthiness of vulnerability detection systems.

Description of the Framework

This section introduces the VISION framework, which improves robustness and interpretability in vulnerability detection. Focused on CWE-20, it uses

LLM-generated counterfactuals to balance training data and boost generalization. Code samples are converted into Code Property Graphs (CPGs) via Joern and embedded for the Devign model. Explanations are generated using the Illuminati explainer, with an interactive module supporting qualitative attribution analysis. Figure E1 shows the full architecture.

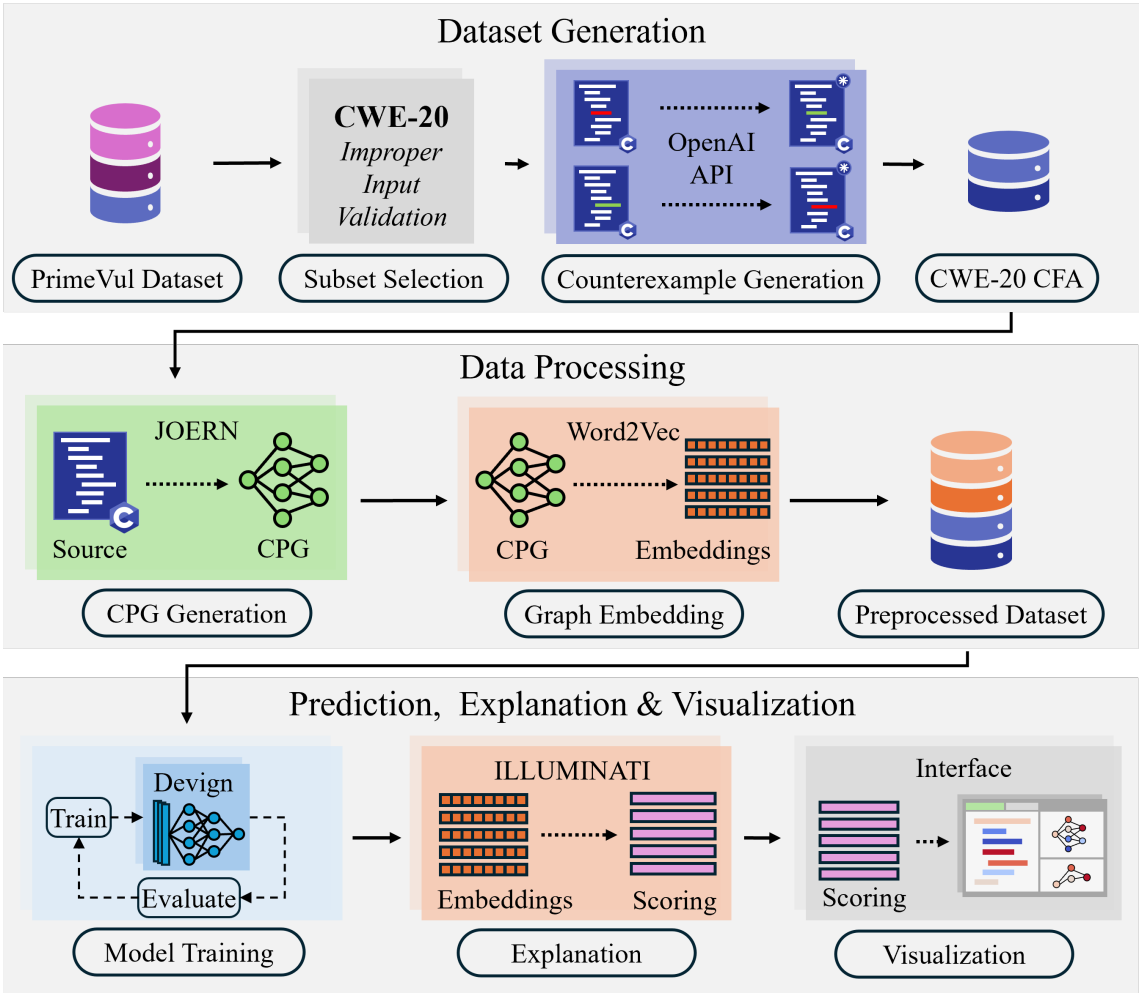


Figure E1: Overview of the VISION framework architecture. Starting from the original source dataset, the end-to-end pipeline includes CWE-20 filtering, counterfactual generation for class balancing, CPG construction, embedding extraction, model training, and explanation generation supported by the visualization module for interpretability.

Dataset selection: CWE-20 Vulnerability

To evaluate the VISION framework under realistic conditions, this work utilizes the PrimeVul dataset [6], a recently released and rigorously curated benchmark for vulnerability detection in source code. PrimeVul offers human-verified labels and strong de-duplication standards, providing a reliable foundation for training and evaluation with reduced label noise and high semantic diversity.

This project focuses specifically on the CWE-20 vulnerability class, known as Improper Input Validation [16], selected based on the following criteria:

1. Semantic clarity. These vulnerabilities often follow well-defined patterns—such as missing boundary checks or unchecked buffers—making them suitable for both automated and manual analysis.

2. Sufficient data. The PrimeVul dataset includes approximately 14,000 CWE-20 instances, allowing robust model training and evaluation without relying on over-sampling or excessive synthetic data.

3. Practical impact. CWE-20 remains one of the most frequently exploited vulnerability types, highlighting the importance of accurate detection in real-world software security [17].

An example illustrating CWE-20 (Figure E2) is the function `validGlxScreen`, which validates whether a given screen index is within bounds. Although it checks for upper limits, it fails to reject negative indices, leading to potential out-of-bounds access. This type of oversight exemplifies the subtle but critical nature of input validation flaws and motivates the need for precise detection mechanisms.

```
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 E2: CWE-20 Improper Input Validation example. The `validGlxScreen` function verifies that the `screen` index does not exceed the number of available screens but neglects to check for negative values. This oversight can lead to invalid array access and exemplifies a typical improper input validation vulnerability.

Counterfactual Generation and Augmentation

A counterfactual is defined as a minimally modified version of a code function whose vulnerability label is reversed relative to the original. These modifications preserve syntactic and semantic correctness while altering the vulnerability condition—transforming a benign sample into a vulnerable one or vice versa. This approach is inspired by counterfactual reasoning in explainable AI [18] and builds on ideas from contrastive learning that expose models to near-boundary examples [19], [20].

In the context of this work, counterfactual examples serve two key purposes: they balance the dataset across classes and guide the model to recognize subtle semantic differences indicative of real vulnerabilities. Unlike traditional augmentation methods that may introduce noise patterns, counterfactuals maintain high fidelity to real-world code structures, supporting stable training and generalization.

Counterfactuals were generated using a prompt-based rewriting strategy via the OpenAI GPT-4o-mini model, leveraging dynamically constructed prompts that include the original code snippet, its label, and CWE-type context. Each function was modified to flip its vulnerability label, either introducing or removing a CWE-20 flaw. An example is shown in Figure E3, where the insertion of non-validated user input in a benign function creates a realistic vulnerability.

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 E3: Example of a counterfactual code pair. The top function is benign, safely calling `net_cmd` with a fixed argument. The bottom version introduces a CWE-20 (Improper Input Validation) vulnerability by substituting the fixed argument with unchecked user input (`user_input`), demonstrating a minimal semantic change that alters the security label.

To construct a balanced training dataset, 14,944 CWE-20 samples were extracted from the PrimeVul dataset, with a strong imbalance toward benign samples. Counterfactuals were then generated from both classes, with problematic or invalid samples removed. The final dataset—CWE-20-CFA—contains 27,556 functions: 13,778 originals and 13,778 counterfactuals, evenly distributed between benign and vulnerable classes (see Table E1).

Table E1: CWE-20 Dataset Filtering and Balancing Summary

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

Base Model for Vulnerability Detection

The VISION framework implements Devign [5] as the base Graph Neural Network architecture for vulnerability classification. Devign is specifically tailored for source code analysis and operates on Code Property Graphs (CPGs) [2], which integrate abstract syntax trees (AST), control flow graphs (CFG), and data flow graphs

(DFG) into a unified representation.

The architecture comprises three main components: (1) a graph embedding layer that encodes node and edge features into latent representations; (2) Gated Graph Recurrent Units (GGRU) that propagate and aggregate information across the graph structure; and (3) a convolutional module that performs graph-level classification by summarizing learned node embeddings.

The Devign model was originally evaluated on large-scale, manually labeled C projects (e.g., Linux Kernel, QEMU, Wireshark, FFmpeg), showing notable performance gains over prior methods. VISION adopts Devign as its baseline due to its demonstrated effectiveness in capturing fine-grained semantic patterns for vulnerability classification.

Visualization Module

To support interpretability and human-in-the-loop analysis, this project integrates a dedicated visualization module designed to inspect individual source code examples and explain the behavior of the trained model through input attributions. The module serves to validate the core hypothesis that counterfactual-based augmentation improves not only accuracy and robustness but also the semantic alignment of model explanations.

The system is built on top of the Illuminati explainer [12], which generates node-level importance scores by extracting minimal and sufficient subgraphs from the GNN model. Predictions are visually encoded: correct classifications are shown in green, while misclassifications appear in red, providing immediate feedback on model performance for human analysis.

Two synchronized views are provided for each function: a highlighted source code display and a graph-based node map. In both views, a continuous color scale is used to indicate the relative importance of tokens or nodes, helping users identify which elements the model considered most influential during prediction.

As illustrated in Figure E4, the visualization reveals how the model's focus shifts between benign and vulnerable function variants, supporting a deeper understanding of decision logic influenced by counterfactual structure.

Experiments and Results

To assess the effectiveness of the VISION framework, a comprehensive evaluation was conducted across a series of training benchmarks composed of varying proportions of original and counterfactual functions. All experiments were based on the CWE-20-CFA dataset and aimed to measure not only predictive accuracy but also robustness to spurious correlations.

Multiple benchmark configurations were created ranging from 100% original to 100% counterfactual data in 10% increments. Each configuration maintained a

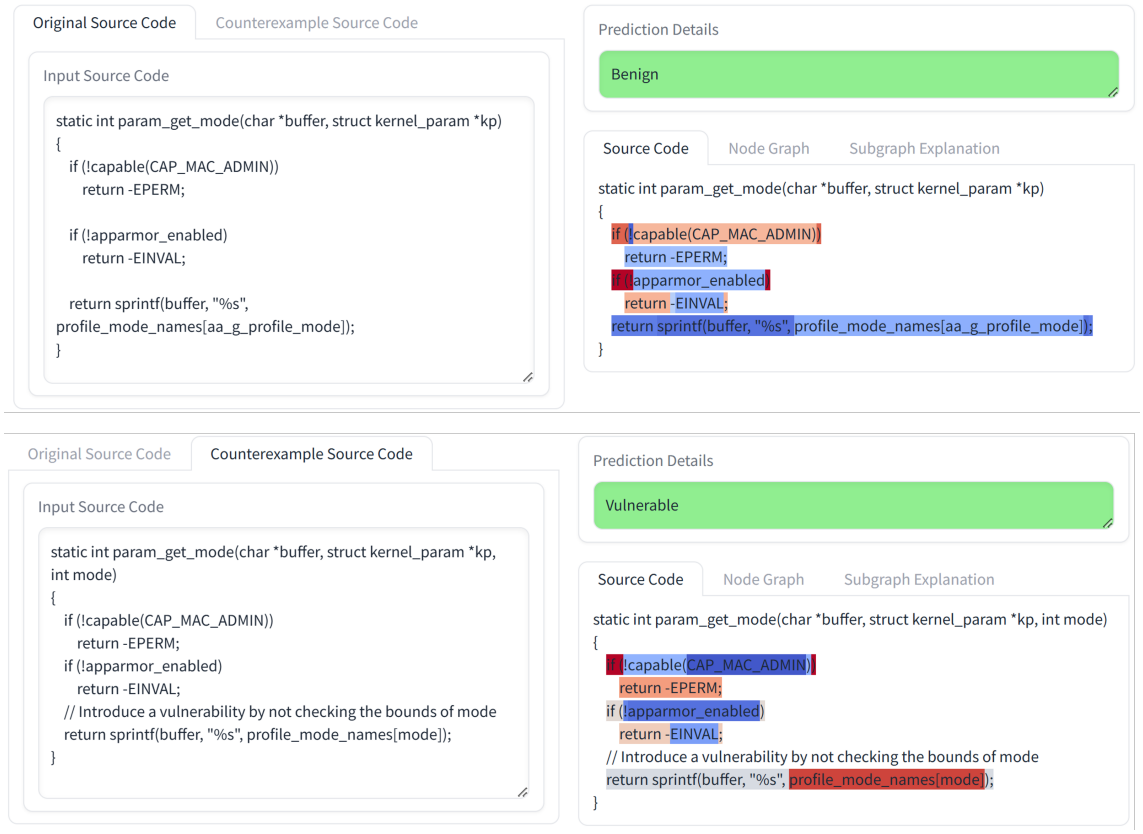


Figure E4: Integrated visualization module displaying model predictions and explanation scores for an original benign function (top) and its vulnerable counterfactual (bottom). Attribution scores are visualized on the right, with red shading indicating token importance. The module facilitates intuitive exploration of attribution changes, revealing how the counterfactual modification impacts both the model's prediction and its reasoning.

constant number of training samples and a perfectly balanced class distribution (benign vs. vulnerable). Dataset splits followed an 80/10/10 ratio for training, validation, and testing, with pairing integrity preserved by assigning each function ID exclusively to one split. The test set remained fixed across all evaluations to ensure consistency and included both the original and counterfactual versions of each function.

Standard metrics—accuracy, precision, recall, and F1-score—were used to evaluate overall performance, while additional metrics such as pairwise accuracy, worst-group accuracy, embedding space analysis, and attribution-based scores were employed to assess robustness and generalization. Results show that performance consistently improves with counterfactual augmentation, peaking around a 50/50 split. The 100% original configuration achieved high precision but suffered from poor recall, indicating overfitting to superficial code patterns. Conversely, the 0% original (fully synthetic) setup degraded performance, confirming that real examples are essential for effective training. These trends validate that a balanced integration of counterfactuals promotes more robust and generalizable learning (see Table E2).

Table E2: Comprehensive evaluation across training splits on the balanced test set. The left section reports accuracy, precision, recall, and F1-score. The right section evaluates robustness, generalization, and explanation quality. Metrics include: P-C (both correct), P-V (both predicted vulnerable), P-B (both predicted benign), and P-R (flipped predictions); WGA (Worst-Group Accuracy, k=4) reflects subgroup robustness. Neighborhood Purity measures class consistency in embedding space. Intra-class Attribution Variance (lower is better) and Inter-class Attribution Distance (higher is better) evaluate explanation consistency and class separability.

Split	Acc	Prec	Rec	F1	P-C	P-V	P-B	P-R	WGA4	Purity	Intra-B	Intra-V	Inter-D
100/0	0.518	1.000	0.036	0.069	4.50	0.00	95.43	0.07	0.0073	0.707	0.01103	0.01027	0.00061
90/10	0.867	0.996	0.737	0.847	74.09	1.38	23.88	0.65	0.7052	0.907	0.01120	0.01035	0.00073
80/20	0.955	0.960	0.948	0.954	91.07	5.44	3.27	0.22	0.8757	0.953	0.01096	0.01046	0.00027
70/30	0.970	0.960	0.980	0.970	94.63	4.86	0.36	0.15	**0.8757**	0.962	0.01109	0.00995	0.00010
60/40	**0.978**	0.961	**0.997**	**0.979**	93.69	6.31	0.00	**0.00**	0.8703	**0.967**	0.01134	0.01030	0.00010
50/50	0.960	0.957	0.962	0.960	**95.79**	**0.44**	**0.00**	3.77	0.8555	0.944	**0.01061**	**0.01030**	**0.00160**
40/60	0.970	**0.998**	0.941	0.969	94.12	1.02	4.50	0.36	0.8092	0.966	0.01122	0.01036	0.00017
30/70	0.951	0.949	0.953	0.951	87.52	8.13	3.85	0.51	0.8266	0.941	0.01101	0.01010	0.00038
20/80	0.930	0.904	0.962	0.932	70.97	27.72	1.02	0.29	0.8497	0.929	0.01144	0.01036	0.00028
10/90	0.919	0.875	0.978	0.924	77.94	20.54	0.65	0.87	0.8152	0.910	0.01103	0.01046	0.00008
0/100	0.799	0.726	0.957	0.826	41.51	57.40	0.65	0.44	0.5030	0.856	0.01122	0.01007	0.00099

Robustness and Spurious Correlation Analysis

Pair-wise accuracy evaluates a model's ability to distinguish between semantically similar code functions with opposite vulnerability labels—typically an original and its counterfactual. A high score indicates that the model is sensitive to subtle, meaningful code edits that alter vulnerability status, rather than relying on superficial patterns. The metric is decomposed into four outcomes: Pair-Correct (P-C), Pair-Vulnerable (P-V), Pair-Benign (P-B), and Pair-Reversed (P-R). Optimal models exhibit high P-C and low values in the other three categories. Results show that the 50/50 original/counterfactual split achieves the highest contrast accuracy (95.79%), confirming that balanced augmentation best supports the model in learning security-relevant distinctions.

Worst-Group Accuracy (WGA) measures a model's weakest performance across automatically discovered subgroups formed by clustering latent embeddings based on code structure and class. It reflects robustness by indicating whether the model overfits to dominant patterns while failing on minority or harder-to-generalize regions. Groups are defined using K-means and intersected with ground-truth labels; WGA is computed as the lowest classification accuracy among sufficiently large groups. Models trained with moderate counterfactual augmentation exhibit the highest and most stable WGA values (¿85%), confirming that counterfactuals enhance subgroup generalization.

Neighborhood Purity assesses how well the learned graph embeddings cluster according to true class labels. It is computed using k-nearest neighbor (kNN) consistency in latent space and serves as an indicator of whether the model organizes representations semantically or relies on shortcut features. Balanced counterfactual integration improves purity, with the 60/40 configuration achieving the highest score, while the 100/0 model shows low purity due to poorly

structured embeddings. Visualizations via t-SNE further confirm that moderately augmented models produce clearer class separation in latent space, suggesting that counterfactuals support more meaningful representation learning (see Figure E5).

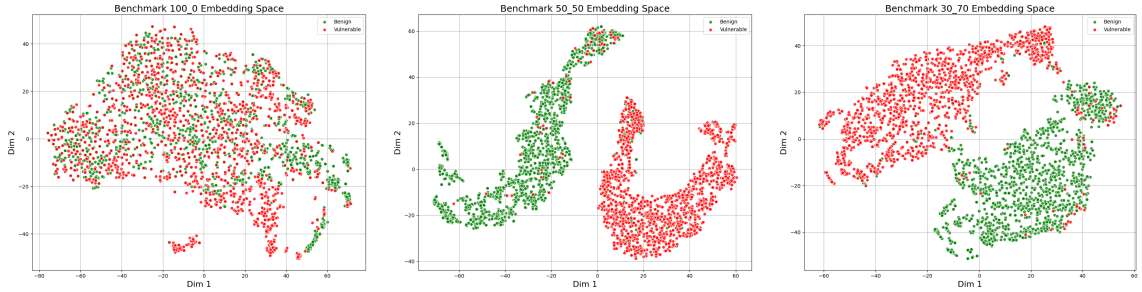


Figure E5: t-SNE projections of graph embeddings for benchmarks 100/0 (left), 50/50 (center), and 30/70 (right). Green points represent benign samples, and red points represent vulnerable samples. These visualizations illustrate how different original/counterfactual splits influence the spatial distribution and separation of vulnerability classes.

The attribution-based metrics evaluate explanation consistency and class separability. Intra-class variance measures how similar the explainer attributions are across samples of the same class—lower variance implies consistent reasoning. Inter-class distance quantifies how distinct the average attribution vectors are between benign and vulnerable samples—higher values suggest clearer conceptual boundaries. Results indicate that intra-class variance remains relatively stable, while inter-class distance peaks in the 50/50 benchmark.

Node Score Dependency analyzes inter-node influence in GNN-based explanations by measuring how importance scores of one node change when another is removed. This yields a dependency matrix that reveals attribution entanglements. In models trained without counterfactuals (100/0), spurious focus is observed on irrelevant tokens, while truly relevant nodes remain uninfluential. In contrast, the 50/50 model exhibits structured dependencies among vulnerability-inducing components, reflecting more accurate and context-aware reasoning (see Figure E6). This metric offers insights into both interpretability and model robustness by identifying fragile or shortcut-driven reasoning paths.

Conclusions

This work presents VISION, a framework for source code vulnerability detection that enhances robustness and interpretability through counterfactual data augmentation. By generating semantically valid code modifications that invert vulnerability labels, the framework exposes Graph Neural Networks to subtle but meaningful patterns, reducing reliance on spurious correlations. To further enhance interpretability, VISION incorporates graph-based explanations through node-level attributions and includes an interactive visualization module to support human-in-the-loop analysis.

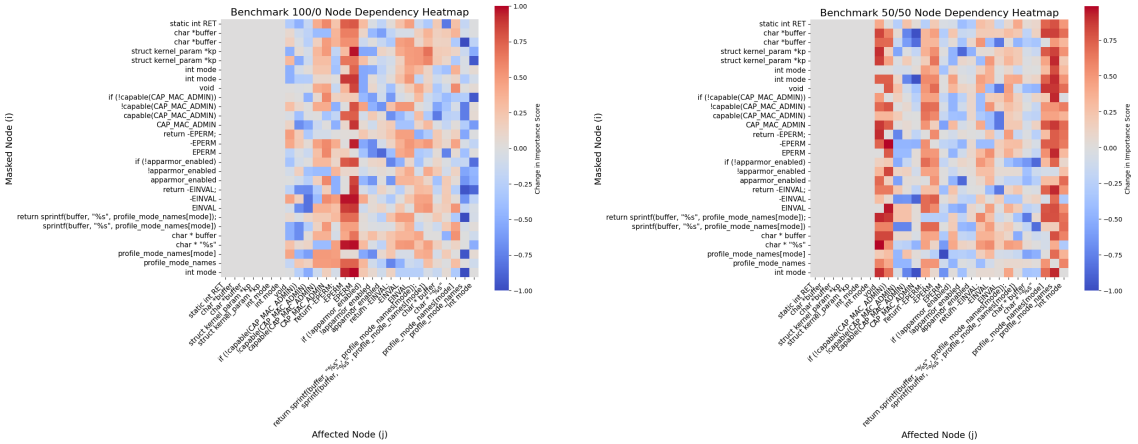


Figure E6: Node Score Dependency heatmaps for the same vulnerable function under two training regimes: Left: Benchmark 100/0 and Right: Benchmark 50/50. Each heatmap shows how masking one node (rows) affects the attribution scores of others (columns). Red indicates increased influence; blue indicates reduced importance. The 100/0 model exhibits high focus on spurious context nodes, while 50/50 shows more meaningful attribution alignment.

The evaluation across multiple benchmarks demonstrates that counterfactual augmentation leads to consistent gains in accuracy, robustness, and explanation quality. Notably, the framework achieves high pairwise accuracy and worst-group accuracy, producing more semantically structured embedding spaces and attribution patterns. These outcomes validate VISION as an effective strategy for mitigating shortcut learning.

However, two limitations currently constrain the generality of the framework. First, the experiments are limited to a single vulnerability class—CWE-20 (Improper Input Validation)—which may restrict applicability to broader security contexts. Second, the use of LLM-generated counterfactuals may occasionally introduce unrealistic or semantically noisy edits.

Future work will focus on extending VISION to additional vulnerability classes and exploring alternative counterfactual generation strategies, such as relying on formal verification. Further efforts will also explore methods to evaluate and enforce semantic correctness of counterfactuals, enabling broader adoption of this framework for building robust and transparent AI systems in cybersecurity.

Bibliography

[1] Boris Chernis and Rakesh Verma. 2018. *Machine Learning Methods for Software Vulnerability Detection.* In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics (IWSPA '18)*, ACM, New York, NY, USA, pp. 31–39. DOI: 10.1145/3180445.3180453.

[2] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. *Modeling and Discovering Vulnerabilities with Code Property Graphs.* In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*, IEEE, pp. 590–604. DOI: 10.1109/SP.2014.44.

[3] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. *The Graph Neural Network Model. IEEE Transactions on Neural Networks*, 20(1): 61–80. DOI: 10.1109/TNN.2008.2005605.

[4] Jingjing Wang, Minhuan Huang, Yuanping Nie, Xiaohui Kuang, Xiang Li, and Wenjing Zhong. 2023. *Fine-Grained Source Code Vulnerability Detection via Graph Neural Networks.* Available at: https://openreview.net/forum?id=S5RYm-9Q4o.

[5] 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.* arXiv preprint arXiv:1909.03496 [cs.SE].

[6] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. *Vulnerability Detection with Code Language Models: How Far Are We?* arXiv preprint arXiv:2403.18624 [cs.SE].

[7] Yuejun Guo and Seifeddine Bettaieb. 2023. *An Investigation of Quality Issues in Vulnerability Detection Datasets.* In *Proceedings of the 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, pp. 29–33. DOI: 10.1109/EuroSPW59978.2023.00008.

[8] Roland Croft, M. Ali Babar, and Mehdi Kholoosi. 2023. *Data Quality for Software Vulnerability Datasets.* arXiv preprint arXiv:2301.05456 [cs.SE].

[9] Wenqian Ye, Guangtao Zheng, Xu Cao, Yunsheng Ma, and Aidong Zhang. 2024. *Spurious Correlations in Machine Learning: A Survey.* arXiv preprint arXiv:2402.12715 [cs.LG].

[10] Samuel J. Bell and Skyler Wang. 2024. *The Multiple Dimensions of Spuriousness in Machine Learning*. arXiv preprint arXiv:2411.04696 [cs.LG].

[11] David Steinmann, Felix Divo, Maurice Kraus, Antonia Wüst, Lukas Struppek, Felix Friedrich, and Kristian Kersting. 2024. *Navigating Shortcuts, Spurious Correlations, and Confounders: From Origins via Detection to Mitigation*. arXiv preprint arXiv:2412.05152 [cs.LG].

[12] Haoyu He, Yuede Ji, and H. Howie Huang. 2023. *Illuminati: Towards Explaining Graph Neural Networks for Cybersecurity Analysis*. arXiv preprint arXiv:2303.14836 [cs.LG].

[13] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. *GNNExplainer: Generating Explanations for Graph Neural Networks*. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*.

[14] Zhaoyang Chu, Yao Wan, Qian Li, Yang Wu, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 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)*, ACM, pp. 389–401. DOI: 10.1145/3650212.3652136.

[15] Deepak Kumar Sharma, Jahanavi Mishra, Aeshit Singh, Raghav Govil, Gautam Srivastava, and Jerry Chun-Wei Lin. 2022. *Explainable Artificial Intelligence for Cybersecurity. Computers and Electrical Engineering*, 103:108356. DOI: 10.1016/j.compeleceng.2022.108356.

[16] MITRE. 2025. *CWE-20: Improper Input Validation*. Available at: https://cwe.mitre.org/data/definitions/20.html. Accessed: 2025-06-03.

[17] MITRE. 2022. *2022 CWE Top 25 Most Dangerous Software Weaknesses*. Available at: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[18] Sandra Wachter, Brent Mittelstadt, and Chris Russell. 2018. *Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR*. arXiv preprint arXiv:1711.00399 [cs.AI].

[19] Divyansh Kaushik, Eduard Hovy, and Zachary C. Lipton. 2020. *Learning the Difference that Makes a Difference with Counterfactually-Augmented Data*. arXiv preprint arXiv:1909.12434 [cs.CL].

[20] Alexis Ross, Ana Marasović, and Matthew E. Peters. 2021. *Explaining NLP Models via Minimal Contrastive Editing (MiCE)*. arXiv preprint arXiv:2012.13985 [cs.CL].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This research aims to enhance software vulnerability detection by addressing two critical limitations of current machine learning models: lack of robustness and limited interpretability. In particular, the goal is to improve model reliability and robustness through counterexample-based data augmentation and to increase transparency via graph-based explanations. These contributions are designed not only to support software developers and security analysts but also to promote broader trust in the use of artificial intelligence for critical cybersecurity tasks.

From a societal perspective, the implications of this work are substantial. Software systems underpin nearly every aspect of modern life—from healthcare and finance to infrastructure and communication. As digital services grow in complexity and scale, the cost of undetected vulnerabilities becomes increasingly severe. High-profile incidents of software exploitation have demonstrated that even minor flaws can lead to massive breaches, financial damage, or systemic failures. Ensuring that machine learning systems can detect such vulnerabilities reliably—and explain their decisions—is essential for the safety and trustworthiness of digital infrastructure.

Technically, the field of AI-driven vulnerability detection has shown promising progress with the use of Graph Neural Networks (GNNs), which learn patterns from structured representations of code. However, these models are often sensitive to spurious correlations, overfitting on superficial code features that do not generalize beyond the training data. At the same time, they typically function as opaque "black boxes," providing little visibility into the reasoning behind their predictions. This undermines their reliability in high-stakes environments and limits their utility for practitioners who need to understand and validate security alerts.

By combining counterfactual data augmentation—which exposes models to semantically meaningful label-inverting examples—with graph-based explainability techniques, this project addresses both limitations in a unified way. The outcome is a model that not only performs better under diverse conditions but also provides interpretable insights into its predictions, helping bridge the gap between automated detection and human decision-making in secure software development.

1.2 Background: The Challenge of Source Code Security

As software systems continue to expand in scale and complexity, ensuring their security has become a central concern in both industry and research. A primary threat vector in this domain is the presence of vulnerabilities in source code, which can be exploited to compromise system integrity, escalate privileges, exfiltrate data, or disable services. Detecting such vulnerabilities early in the development cycle is crucial to minimizing security risks and reducing remediation costs.

Traditional approaches to vulnerability detection—such as static analysis, dynamic analysis, and symbolic execution—rely heavily on manually crafted rules or predefined security patterns. While these methods have been widely adopted in development pipelines, they often suffer from limited scalability, high false-positive rates, and difficulty in generalizing to previously unseen vulnerabilities [Din+24]. Moreover, they tend to operate on surface-level syntactic structures, lacking the deeper semantic understanding required to capture complex code behaviors and subtle security flaws.

To address these limitations, recent research has explored the use of machine learning, and more specifically, Graph Neural Networks (GNNs) [Sca+09], as a powerful alternative. GNNs are well-suited for this task because source code can naturally be represented as Code Property Graphs (CPGs) [Yam+14], such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). These structures encode relationships between code tokens, control paths, and variable interactions—enabling models to reason about code behavior in a more holistic and context-aware manner (see Figure 1.1).

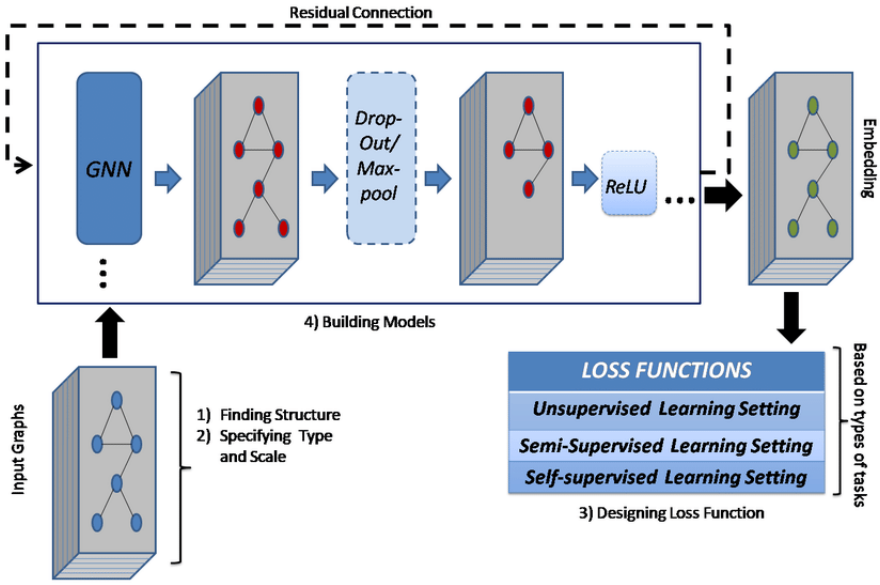


Figure 1.1: Overview of a typical Graph Neural Network (GNN) architecture, illustrating the main components such as input graphs, GNN layers, pooling, activation functions, and embedding generation. Adapted from [WP21].

Despite their potential, GNN-based models for vulnerability detection face several key challenges:

- Robustness: Public vulnerability datasets often contain noisy labels, imbalanced class distributions, and duplicated or near-duplicate functions. These issues can cause models to learn superficial correlations—e.g., associating vulnerability with specific variable names or patterns—rather than true semantic indicators of unsafe behavior. As a result, models may achieve high accuracy on benchmark datasets but fail to generalize to real-world codebases.
- Lack of Explainability: Like many deep learning models, GNNs typically behave as black boxes, offering little insight into why a particular piece of code is classified as vulnerable. This lack of transparency limits their practical applicability in security workflows, where human analysts need to understand and verify model outputs before taking action.
- Dataset Limitations: High-quality labeled datasets are scarce, and those that exist often contain class imbalance, inconsistent labeling, and insufficient coverage of real-world code patterns. This restricts the training and evaluation of models, especially in security contexts where even small biases can lead to critical errors.

Overcoming these challenges requires not only algorithmic innovation but also a deeper integration between robust learning strategies and interpretability techniques. This project contributes to that effort by proposing a framework that jointly improves both model robustness and transparency—laying the groundwork for more reliable and trustworthy vulnerability detection systems.

1.3 Proposed Solution

To address the challenges outlined in the previous section, this work proposes **VISION** (*Vulnerability Identification and Spuriousness mitigation via counterfactual augmentatION*)—a unified framework designed to improve both the robustness and interpretability of GNN-based vulnerability detectors. VISION introduces a novel data augmentation strategy based on counterfactual examples, enabling the model to learn from subtle, meaningful changes in source code that directly impact its vulnerability status. This encourages the detection of true vulnerability patterns while reducing the model's reliance on superficial or spurious features.

The central idea of VISION is to generate paired code samples that are minimally different in syntax but opposite in label—transforming, for example, a benign function into a vulnerable one by introducing an input validation flaw, or vice versa. These counterfactuals are generated automatically using a prompt-based rewriting strategy powered by large language models (LLMs). Each prompt includes the function's original source code and metadata such as its vulnerability label and CWE type, guiding the model to apply controlled, semantically consistent edits.

Vulnerable code (CWE-416: Use After Free): unguarded free of `rc_urb`

```
static void snd_usb_mixer_free(struct usb_mixer_interface *mixer){
    kfree(mixer->id_elems);
    if (mixer->urb) {
        kfree(mixer->urb->transfer_buffer);
        usb_free_urb(mixer->urb);
    }
    usb_free_urb(mixer->rc_urb); // unguarded free causes null pointer deref
    kfree(mixer->rc_setup_packet);
    kfree(mixer);
}
```

Counterfactual fix: guard on `rc_urb` before free

```
static void snd_usb_mixer_free(struct usb_mixer_interface *mixer){
    kfree(mixer->id_elems);
    // only free rc_urb if it was ever allocated
    if (mixer->rc_urb) {
        usb_free_urb(mixer->rc_urb);
    }
    kfree(mixer->rc_setup_packet);
    kfree(mixer);
}
```

Figure 1.2: Top: vulnerable snippet with an unguarded call to `usb_free_urb(mixer->rc_urb)` (CWE-416). Bottom: minimal counterfactual insertion of `if (mixer->rc_urb)` removes the bug. This demonstrates a spurious correlation risk: if the training examples share a pattern, the model may learn to flag that it as "vulnerable," rather than recognizing the true root cause.

A compelling example of the spurious correlation phenomenon is shown in Figure 1.2. The top listing contains a genuine CWE-416 (*Use After Free*) vulnerability due to an unguarded call to `usb_free_urb(mixer->rc_urb)`. The bottom listing is a benign variant that introduces a simple null-check before the same call. Despite the clear semantic difference, a model trained without proper counterfactuals might still flag the safe version as vulnerable—if it has learned to associate the mere presence of `usb_free_urb` with buggy behavior. This highlights a core weakness in current learning-based detectors: they often rely on superficial code patterns instead of understanding the true security implications of control flow and logic. Counterfactual augmentation directly addresses this by presenting the model with paired examples that differ minimally in syntax but substantially in semantics.

To support this approach, a new benchmark dataset is introduced: CWE-20-CFA. It is derived from PrimeVul [Din+24], a high-quality vulnerability dataset, by filtering all examples labeled with CWE-20 (Improper Input Validation) [MIT25]—a common and high-impact vulnerability category. The original data exhibited a significant class imbalance (with benign samples vastly outnumbering vulnerable ones), which was resolved through the counterfactual augmentation process. The final dataset contains 27,556 samples, evenly split between benign and vulnerable classes, and includes both original functions and their validated counterfactual counterparts.

The VISION framework leverages the Devign architecture [Zho+19], a GNN model well-suited for vulnerability detection in source code, as its predictive core. Code samples are first converted into CPGs using the Joern [Whi25] analysis tool,

which combines abstract syntax, control flow, and data flow representations. These graphs are then encoded into embeddings and passed through the GNN for training.

Beyond classification, VISION also integrates the Illuminati [HJH23] explainer to provide graph-based attributions—highlighting the most influential code nodes and tokens that drive model decisions. An interactive visualization module complements this functionality by allowing human analysts to inspect, compare, and verify predictions at both the source code and graph levels. An example of the visualization interface is shown in Figure 1.3, where node attributions are rendered over both the original and counterfactual function graphs. This allows users to visually inspect which regions of the code contributed most to the model's prediction, improving interpretability and trust in the system.

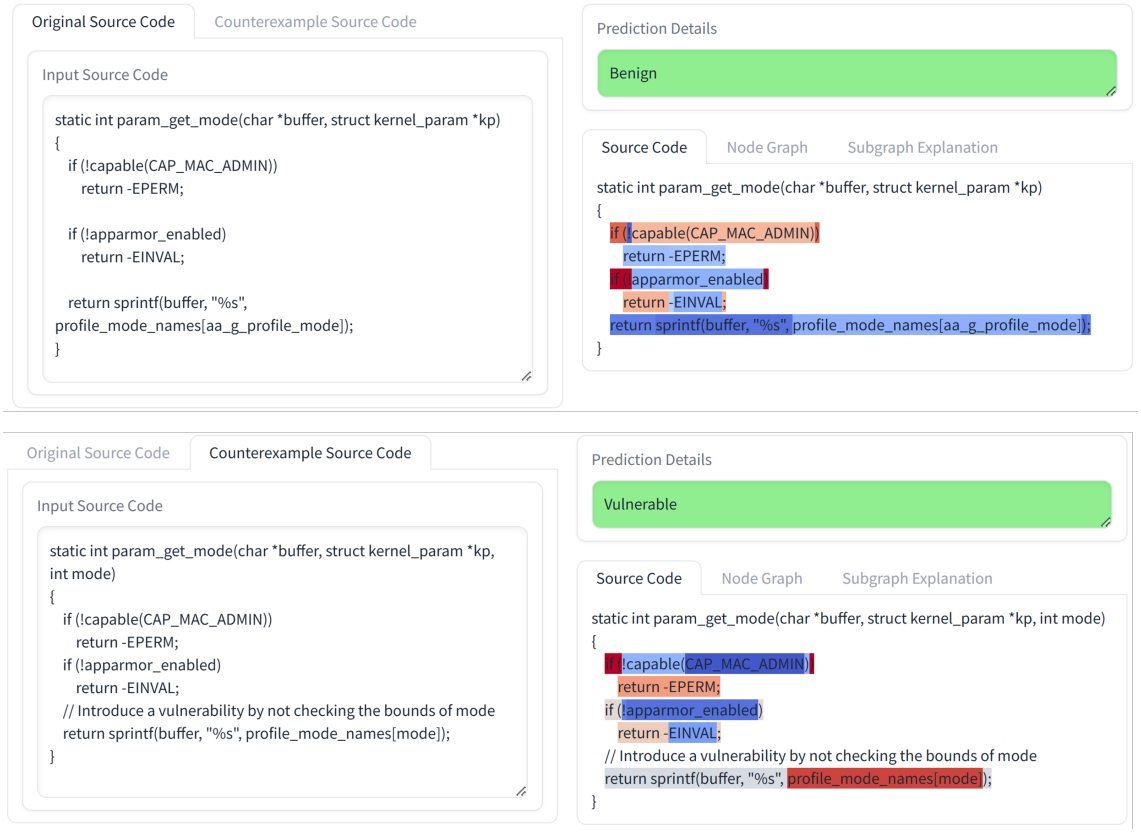


Figure 1.3: Screenshot of the VISION visualization module. The interface shows the original function (top) and its counterfactual (bottom), with graph-based node attributions highlighted. Red nodes indicate high importance for the vulnerability prediction. This tool enables human-in-the-loop analysis by linking predictions with semantically relevant code structures.

The effectiveness of VISION was validated through a series of experiments using benchmark splits with varying ratios of original and counterfactual data. Results demonstrate substantial improvements in both predictive performance and explanation quality. Notably, pairwise accuracy increases from 4.5% to 95.8%, and worst-group accuracy rises from 0.7% to over 85%—highlighting the impact of counterfactual augmentation in reducing shortcut learning and enhancing generalization.

1.4 Conclusions

This chapter has introduced the motivation, background, and central objectives of this research, situating the problem of vulnerability detection within both a societal and technical context. As software continues to underpin critical infrastructures and digital services, the importance of developing robust and interpretable vulnerability detection systems becomes increasingly evident. Traditional detection tools fall short when faced with the scale, semantic diversity, and subtlety of modern codebases—often relying on rigid heuristics or superficial patterns that fail to generalize.

To address these limitations, this work proposes VISION, a unified framework that enhances the performance and transparency of Graph Neural Network (GNN)-based vulnerability detection. By integrating counterfactual data augmentation with graph-based explanation techniques, the framework encourages models to focus on truly discriminative features while providing interpretable insights into their decisions. The use of large language models to generate minimal label-flipping edits allows for a systematic augmentation process that exposes the model to subtle semantic boundaries, helping to reduce spurious correlations.

To summarize, the key contributions of this work are:

- A **novel counterfactual augmentation strategy** that uses Large Language Models (LLMs) to improve robustness by presenting models with paired code examples that differ minimally in syntax but meaningfully in vulnerability semantics.
- **Empirical validation** on a realistic and challenging benchmark focused on CWE-20 vulnerabilities, demonstrating significant improvements in robustness and generalization without requiring external data sources.
- The construction of a **new benchmark dataset CWE-20-CFA**, created by augmenting existing CWE-20 samples with validated counterfactuals, enabling more balanced and insightful training and evaluation.
- An **interactive visualization module** based on graph-based explanations, allowing users to qualitatively assess model behavior and observe attribution patterns aligned with semantic vulnerability cues.

The following chapters delve into the technical details, prior literature, implementation, and evaluation of the VISION framework.chapter 2 presents the related work in vulnerability detection and explainable machine learning; chapter 3 describes the design and methodology behind VISION; chapter 4 details the experimental setup and results; and chapter 5 concludes the thesis with a reflection on limitations and future directions.

Chapter 2

Related Work

The task of detecting vulnerabilities in source code intersects multiple areas of research, including software security, machine learning, graph representation learning, and explainable AI. This chapter reviews the key developments and ongoing challenges across these domains, highlighting the limitations of traditional analysis tools, the growing role of deep learning—especially Graph Neural Networks (GNNs) [Sca+09]—and the emerging importance of interpretability and robustness. Special attention is given to recent advancements in data augmentation, counterfactual reasoning, and benchmark design, which inform the design of the VISION framework.

2.1 Vulnerability Detection in Source Code

Detecting software vulnerabilities has long been a central goal in cybersecurity, as flaws in source code can be exploited to compromise systems, access sensitive data, or disrupt services. Over time, a wide range of approaches has emerged, from rule-based static and dynamic analysis tools to more recent learning-based methods. This section reviews the evolution of vulnerability detection techniques, highlighting the strengths and limitations of traditional tools, and tracing the shift toward data-driven approaches that leverage classical machine learning, deep learning, and transformer-based models.

2.1.1 Static and Dynamic Analysis Tools

Traditional approaches to software vulnerability detection have long relied on static and dynamic analysis techniques. These methods form the foundational layer of software security assessment, each offering distinct advantages and facing specific limitations.

Static Analysis

Static analysis involves examining source code without executing it, aiming to identify potential vulnerabilities, coding errors, and deviations from coding standards early in the development lifecycle. Tools such as SonarQube, Cppcheck, and CodeSonar are widely used in the industry for this purpose [Jha25]. These tools

can detect issues like buffer overflows, null pointer dereferences, and use-after-free errors by analyzing the code's structure and syntax.

Dynamic Analysis

Dynamic analysis, in contrast, involves evaluating a program during its execution. This approach can uncover vulnerabilities that manifest only at runtime, such as memory leaks, race conditions, and improper input validation. Dynamic Application Security Testing (DAST) tools simulate real-world attack scenarios to identify potential security flaws in running applications [OWA25].

While dynamic analysis provides valuable insights into a program's behavior under actual operating conditions, it has its limitations. It can only assess code paths that are executed during testing, potentially missing vulnerabilities in untested paths. Moreover, setting up a realistic testing environment can be complex and resource-intensive, and dynamic analysis may not be feasible in early development stages where the application is not yet fully functional.

Complementary Use and Limitations

Both static and dynamic analysis tools are essential in a comprehensive security strategy, offering complementary insights. Static analysis is effective for early detection of potential issues, while dynamic analysis provides a deeper understanding of runtime behavior. However, neither approach is foolproof. Static analysis may miss context-dependent vulnerabilities, and dynamic analysis may not achieve complete code coverage. Furthermore, both methods can generate false positives and negatives, necessitating manual review and validation.

These limitations underscore the need for more advanced techniques in vulnerability detection. Machine learning and, more recently, deep learning approaches have emerged to address these challenges by learning complex patterns from large codebases, aiming to improve detection accuracy and reduce false positives.

2.1.2 Classical Machine Learning Approaches

As a response to the limitations of static and dynamic analysis, classical machine learning (ML) techniques began to emerge in the early 2000s as a promising alternative for automated vulnerability detection [CV18; Ala+25]. These methods aimed to move beyond hand-crafted rules by learning patterns from labeled examples of vulnerable and non-vulnerable code.
Most classical ML approaches rely on manually extracted features such as token frequencies, control structures, or syntactic patterns derived from abstract syntax trees (ASTs) or control flow graphs (CFGs). For example, support vector machines (SVMs), decision trees, and logistic regression models have been trained on token-level features to predict the likelihood of a code snippet being vulnerable. Vulture [Neu+07] used import frequency analysis to predict vulnerable components (see Figure 2.1), while other works explored the use of n-gram models and bag-of-words representations to capture local syntax patterns [Mou+24].

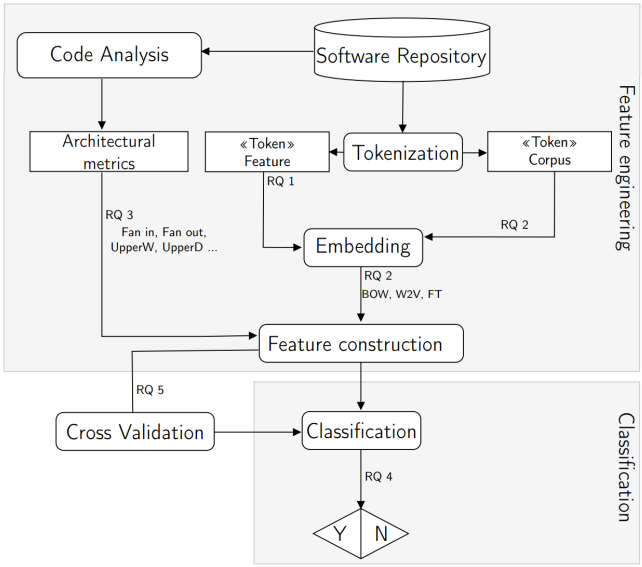


Figure 2.1: Overview of the vulnerable code detection process described in the Vulture paper [Neu+07]. A software repository is used to train a token vocabulary and embeddings (e.g., Word2Vec). Source code is then transformed into token vectors and combined with architecture metrics to feed a supervised classifier, which is trained with in-domain or cross-domain vulnerability labels.

These techniques brought scalability and automation to the vulnerability detection process but were still fundamentally constrained by the quality of feature engineering. Their ability to generalize across different projects, languages, or vulnerability types was limited, and they often struggled to model the complex, structured nature of code semantics. Additionally, most classical ML methods treated code as flat text or sequences, lacking awareness of hierarchical structures and long-range dependencies critical to understanding real-world vulnerabilities.

Despite these limitations, classical ML approaches laid the groundwork for deeper integration of data-driven techniques in software security and opened the door to the adoption of more expressive models, such as deep neural networks and, later, graph-based architectures.

2.1.3 Deep Learning and Neural Models

The introduction of deep learning into the field of software vulnerability detection marked a significant shift toward models capable of automatic feature extraction and greater representational power. Unlike classical machine learning methods, deep learning models can learn complex patterns from raw or lightly preprocessed code without requiring manually engineered features. This flexibility enables the capture of deeper semantic relationships and long-range dependencies, which are critical for identifying subtle and context-dependent vulnerabilities.

Early applications of deep learning in this area focused on sequential representations of code, often using models originally developed for natural language processing (NLP). Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM)

networks were used to model the token sequences of code functions, capturing sequential dependencies in variable usage, API calls, and control structures. Ziems and Wu (2021), for example, applied LSTMs to learn sequential patterns indicative of vulnerabilities across large codebases [ZW21].

Convolutional Neural Networks (CNNs) were also adapted for code analysis, particularly for their ability to detect local syntactic or structural patterns. VulCNN [Wu+22] demonstrated that convolutional architectures could efficiently learn vulnerability patterns in function-level source code, drawing analogies between token arrangements in code and spatial arrangements in image data. An overview of the VulCNN architecture is illustrated in Figure 2.2.

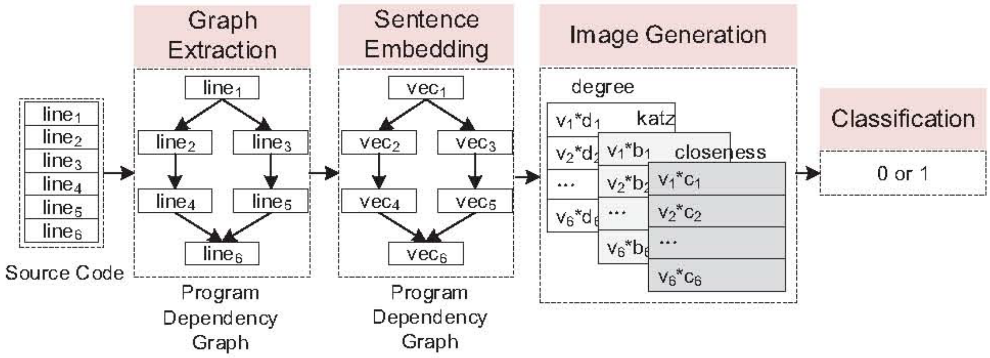


Figure 2.2: Overview of the VulCNN pipeline [Wu+22]. It consists of four stages: (1) extraction of a program dependency graph from source code, (2) sentence-level embedding of code lines as graph nodes, (3) generation of an image by weighting vectors using centrality metrics, and (4) CNN-based vulnerability classification.

With the advent of pretrained transformer models, such as CodeBERT [Fen+20] and GraphCodeBERT [Guo+21], deep learning for code analysis took another leap forward. These models leveraged vast corpora of code to learn high-quality embeddings via masked language modeling, fine-tuned later for tasks such as function classification, defect prediction, or vulnerability detection. Their strength lies in modeling global context and token relationships through self-attention mechanisms, allowing them to understand complex code logic and dependencies beyond what RNNs or CNNs could capture.

Despite these advances, deep learning models—especially sequence-based ones—still face important limitations. Treating code purely as text or sequences neglects the graph-structured nature of programming languages, such as syntax hierarchies and data/control flow relationships. Moreover, deep models often suffer from low interpretability and are prone to overfitting on dataset-specific patterns, especially when trained on imbalanced or noisy benchmarks. These challenges paved the way for Graph Neural Networks (GNNs), which naturally align with the structural properties of source code and are increasingly adopted in vulnerability detection research.

2.1.4 Transformer-Based Models for Code

Transformer-based models have significantly advanced the field of software analysis by enabling pretraining on large-scale code corpora and learning contextual representations that capture both syntax and semantics. Initially developed for natural language processing, transformers leverage self-attention mechanisms to model global dependencies across sequences. In the software domain, these capabilities translate well to capturing long-range interactions in code, such as dependencies between variable definitions and usage, or relationships between function calls and control structures.

CodeBERT [Fen+20] was among the first pretrained models to adapt the BERT architecture for source code. It was trained on a large multilingual corpus of programming and natural language pairs, enabling joint modeling of documentation and implementation. CodeBERT demonstrated strong performance on a range of tasks, including function classification, code search, and defect prediction, and quickly became a widely used foundation model in the code intelligence community.

Building on this, GraphCodeBERT [Guo+21] introduced an improved architecture that incorporates data flow graphs (DFGs) during pretraining (see Figure 2.3), thereby explicitly encoding program semantics. This extension improved the model's ability to reason about code behavior, such as how variables propagate through a function, leading to performance gains on tasks that require deeper semantic understanding.

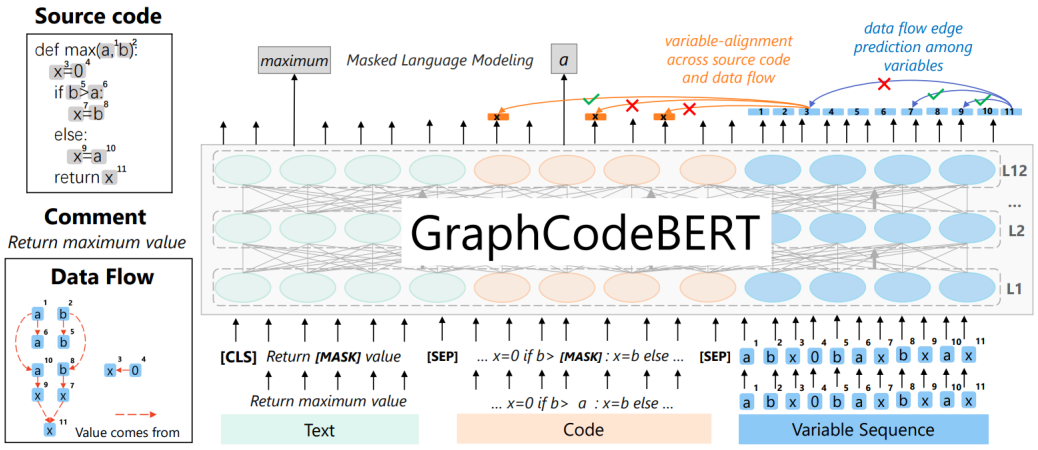


Figure 2.3: Pre-training process of GraphCodeBERT [Guo+21]. The model takes source code paired with comments and corresponding data flow as input. It is trained using masked language modeling along with two structure-aware tasks: (1) predicting the origin variable of a use (orange arrows), and (2) predicting data flow edges between variables (blue arrows).

Other models, such as PLBART [Ahm+21], CodeT5 [Wan+21], and CodeGen [Nij+23], further explore multi-task learning and generative capabilities by combining code summarization, translation, and generation with defect and

vulnerability detection. These models benefit from massive pretraining on code repositories like GitHub and show generalization across languages and tasks.

Large Language Models (LLMs) have also been directly applied to cross-language and general-purpose vulnerability detection, showing competitive or superior performance in recent studies [SAE24; ZZL24]
Despite their success, transformer-based models face limitations when applied to vulnerability detection. Most notably, they still treat source code as linear sequences of tokens, which can obscure the rich structural dependencies. Additionally, they often lack explicit explainability and can overfit to statistical patterns that are not semantically meaningful. As a result, researchers have increasingly turned to graph-based models, which offer a more faithful representation of program structure and can be extended with explanation techniques to improve trust and interpretability in high-stakes security contexts.

2.2 Graph Neural Networks for Code Representation

As programming languages are inherently structured and hierarchical, source code can be naturally represented as graphs—capturing not only the syntax but also the flow of control and data. Graph Neural Networks (GNNs) [Sca+09] have emerged as a powerful tool for leveraging this structure, enabling models to reason about relationships between code elements in a way that goes beyond token-level or sequential representations. This section reviews the fundamental graph representations used in software analysis and explores GNN architectures that have been successfully applied to code understanding and vulnerability detection.

2.2.1 Source Code as Graphs

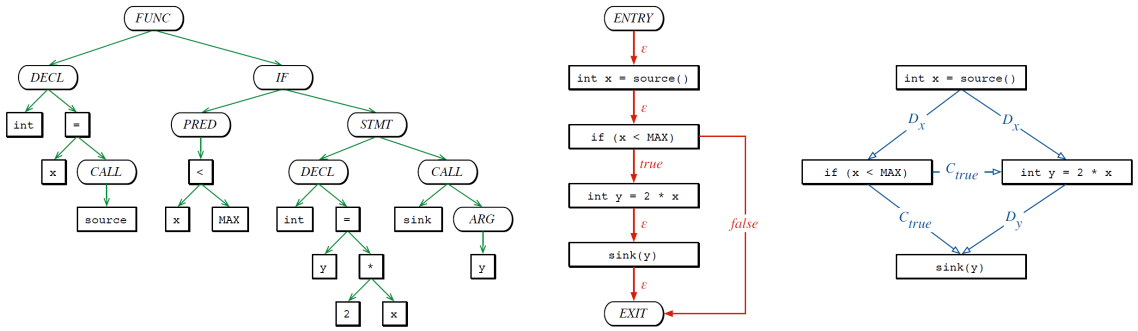
Programming languages, by design, follow strict syntactic and semantic rules, making source code a naturally structured input. Over decades, the fields of program analysis and compiler design have developed multiple intermediate representations to reason about the structure, behavior, and properties of code. These representations not only support optimization and debugging but also serve as a foundation for machine learning models applied to software security.

```
1 void foo()
2 {
3     int x = source();
4     if (x < MAX)
5     {
6         int y = 2 * x;
7         sink(y);
8     }
9 }
```

Figure 2.4: Illustrative code snippet used for Code Property Graph (CPG) explanation. Adapted from [Yam+14].

This section focuses on three core representations—Abstract Syntax Trees (ASTs),

Control Flow Graphs (CFGs), and Program Dependence Graphs (PDGs)—which are fundamental to graph-based vulnerability detection. A running code example is depicted in Figure 2.4.



(a) Abstract syntax tree (AST) (b) Control flow graph (CFG) (c) Program dependence graph (PDG)

Figure 2.5: Illustration of three core code representations for a sample function: (a) Abstract Syntax Tree (AST) captures syntactic structure, (b) Control Flow Graph (CFG) models execution order and branching, (c) Program Dependence Graph (PDG) encodes data and control dependencies. Adapted from [Yam+14].

Abstract Syntax Trees (ASTs)

Abstract Syntax Trees are the earliest and most syntactic representation of source code, generated directly from parsing. ASTs encode the hierarchical nesting of statements and expressions but abstract away surface-level syntactic elements like punctuation or keywords. In an AST, internal nodes represent operators (e.g., assignments, function calls) while leaves correspond to variables, constants, or function names.

As seen in Figure 2.4(a), the AST for the code snippet structurally organizes the assignment, predicate, and function call statements, capturing how program constructs are composed. ASTs are useful for syntax-based pattern matching and code transformations but are limited when deeper analysis—such as variable influence or execution paths—is required. Notably, they do not capture control flow or data dependencies between different parts of the program.

Control Flow Graphs (CFGs)

Control Flow Graphs model the execution order of a program by representing statements and predicates as nodes, with directed edges indicating the possible paths the program may take during runtime. Predicate nodes include branching based on true or false evaluations, enabling explicit modeling of conditional and loop structures.

Figure 2.4(b) shows the CFG of the same example code. It captures whether the call to `sink(y)` is conditionally executed, depending on the value of `x`. CFGs are particularly useful in program understanding, reverse engineering, and vulnerability detection tasks that depend on execution semantics. However, CFGs do not express

how data moves through the program, making them insufficient alone for reasoning about vulnerabilities such as taint propagation or improper data use.

Program Dependence Graphs (PDGs)

Program Dependence Graphs, introduced by Ferrante et al. [FOW87], unify both control and data dependencies within a single representation. PDGs contain two edge types: data dependencies, which connect definitions and uses of variables, and control dependencies, which represent how predicate outcomes influence the execution of statements.

Figure 2.4(c) illustrates how PDGs explicitly encode the influence of x on y and the dependency of `sink(y)` on the condition `x < MAX`. Unlike CFGs, PDGs do not preserve execution order but allow reasoning over semantic dependencies, making them ideal for tasks such as slicing, taint analysis, and vulnerability detection.

Together, ASTs, CFGs, and PDGs form the structural foundation of modern graph-based learning methods for source code. Their combination—often realized in Code Property Graphs (CPGs)—enables rich, multi-view representations that support robust and interpretable learning.

2.2.2 GNN Architectures for Code

Graph Neural Networks (GNNs) [Sca+09] have emerged as a powerful class of models capable of learning from relational and structured data. In the context of source code analysis, GNNs are particularly well-suited due to their ability to process program representations—such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs)—where nodes represent code elements and edges represent syntactic or semantic relationships. These networks enable deep learning models to reason over both local and global code structures—something that traditional sequential models like RNNs or CNNs struggle to achieve.

GNN Fundamentals in Code Analysis

At a high level, GNNs operate by iteratively propagating information across a graph structure. Each node aggregates features from its neighbors using a message-passing mechanism, gradually updating its own representation through multiple layers. This paradigm is naturally applicable to program graphs, where the goal is to compute meaningful embeddings for functions, code blocks, or entire graphs that reflect vulnerability-relevant semantics.

Early applications of GNNs to code analysis focused on tasks such as code summarization, variable misuse detection, and program classification [Li+18]. These tasks benefit from the GNN's ability to model both short- and long-range dependencies among code elements, particularly when working with heterogeneous graphs composed of multiple edge types and semantic roles.

2.2.3 Devign: A GNN Model for Vulnerability Detection

A landmark in this field is Devign [Zho+19], a GNN-based model specifically designed for source code vulnerability detection. Devign introduced a graph-centric approach that captures both syntax and semantic features by operating over Code Property Graphs (CPGs). These graphs combine multiple views of code—including ASTs, CFGs, and DFGs—into a unified graph representation, thereby providing a comprehensive view of program structure.

Architecture

Devign's architecture consists of three core components (see Figure 2.6:

- **Graph Embedding Layer**: This layer encodes code tokens and edge types into continuous vector representations. The input graph, derived from a CPG, includes multiple edge types representing different code relationships (e.g., `AST_CHILD`, `CFG_NEXT`, `DFG_READS`), which are embedded and processed jointly.
- **Gated Graph Recurrent Unit (GGRU)**: This component is responsible for propagating information through the graph. It captures the flow of semantics between code elements via gated recurrent message-passing, allowing node embeddings to evolve based on their neighbors. The GGRU structure is particularly effective for modeling long-range dependencies and cyclic graphs found in real-world code.
- **Graph-level Convolution and Pooling**: After message passing, Devign uses global pooling to summarize node representations into a fixed-size vector. This embedding is passed through dense layers for binary classification, where the output predicts whether a given function is vulnerable or benign.

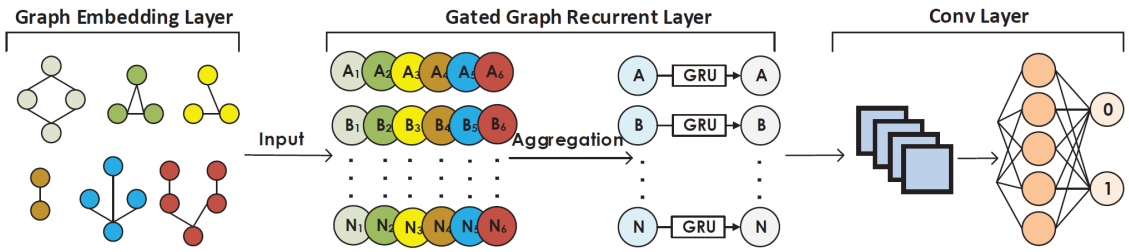


Figure 2.6: Overview of the Devign model architecture as proposed by Zhou et al. [Zho+19]. The model processes graph representations of source code through an embedding layer, gated graph recurrent units (GGRU), and a global pooling mechanism to produce vulnerability predictions. Multiple graph views such as AST, CFG, and DFG are unified in the input graph.

Results and Achievements of the Devign Model

Devign represents a significant advancement in automated vulnerability identification by leveraging graph neural networks (GNNs) to learn from comprehensive program semantics. Evaluated on a large-scale, manually labeled

dataset consisting of real-world open-source C projects—including FFmpeg, QEMU, Wireshark, and the Linux Kernel—Devign demonstrated substantial improvements over prior state-of-the-art models. Specifically, it achieved an average of 10.51% higher accuracy and 8.68% higher F1 score compared to baseline approaches such as CNNs, RNNs, and traditional machine learning models. The introduction of Devign's novel Conv module further contributed to performance, yielding an additional average gain of 4.66% in accuracy and 6.37% in F1 score, underscoring its effectiveness in extracting vulnerability-relevant features from graph-structured code representations.

Beyond outperforming deep learning and static analysis baselines, Devign proved robust across a variety of settings. It maintained strong results even on imbalanced datasets that mirror real-world conditions, surpassing well-known static analyzers by an average of 27.99% in F1 score. Moreover, Devign showed practical utility in identifying newly disclosed vulnerabilities: when tested on 40 recent CVEs from the evaluated projects, it achieved an average accuracy of 74.11%, indicating its potential for discovering zero-day vulnerabilities in complex, real-world codebases. These achievements have established Devign as a foundational architecture in the field, inspiring subsequent research on GNN-based vulnerability detection and setting a new benchmark for effectiveness and generalizability in machine learning for software security.

Advantages and Limitations of Devign

One of Devign's key advantages is its capacity to integrate multiple views of code in a single representation, offering a richer signal for vulnerability classification. Its design aligns well with the underlying structure of programming languages and does not require manual feature engineering, making it broadly applicable to diverse codebases.

However, Devign also exhibits some limitations. The model is highly dependent on the quality and structure of the input graphs, which in turn depend on the accuracy of static analysis tools such as Joern. Furthermore, as a conventional GNN, it can suffer from oversmoothing—where node embeddings become indistinguishable in deeper networks—and may overfit to dataset-specific spurious patterns if not properly regularized or trained on diverse data. Another limitation is the lack of native interpretability, which makes it challenging to understand which code components contribute most to the model's predictions.

2.2.4 Strengths and Limitations of GNNs in Security Tasks

Among their advantages, GNNs offer **context-aware learning**, as node embeddings are iteratively updated through message passing, allowing each node to gather information from its neighbors. This mechanism is particularly useful in code analysis, where vulnerabilities are often determined by dependencies between variables, control branches, and function calls scattered throughout a program. GNNs can also handle heterogeneous graphs with diverse edge types, making them adaptable to richly structured inputs like Code Property Graphs (CPGs). As a result, they outperform traditional sequential models (e.g., RNNs or CNNs) on

tasks where structure and interaction matter, such as vulnerability detection, clone detection, and program classification.

GNNs have demonstrated success in various security-related tasks, such as predicting vulnerabilities, detecting malware variants, and classifying buggy code snippets [Zho+19; Li+18]. Studies have shown that GNN-based models can outperform traditional neural networks when the input is graph-structured, achieving better generalization and capturing more complex patterns of code behavior.

Nevertheless, GNNs exhibit several important limitations in security-sensitive applications. A key challenge is their susceptibility to spurious correlations, where models learn to associate irrelevant features—such as naming conventions or formatting artifacts—with vulnerability labels due to patterns in biased or duplicated training data. This issue is particularly pronounced in noisy or imbalanced datasets, where models may overfit to superficial characteristics instead of learning true vulnerability semantics [Din+24]. Spurious correlations can lead to brittle generalization, especially when evaluating on out-of-distribution samples.

GNNs are also known to suffer from oversmoothing, where node representations converge to similar values after multiple layers of message passing. In the context of source code, this effect can cause semantically distinct code regions to become indistinguishable in latent space, impairing the model's ability to differentiate fine-grained program behaviors. This can limit performance on longer functions or deeply nested structures, where contextual precision is critical.

A further concern is the lack of **interpretability**. GNNs often behave as black boxes, making it difficult to understand which parts of a program led to a particular vulnerability prediction. While various explanation methods have been proposed—such as GNNExplainer [Yin+19], PGM-Explainer [VT20], and SHAP [LL17]-based adaptations—the field of explainable GNNs remains relatively nascent in the context of software security.

Finally, **robustness** is an increasingly important area of concern. Recent studies show that GNNs are vulnerable to structural perturbations, where small changes in the graph (e.g., adding or deleting edges or nodes) can significantly alter the model's prediction (adversarial attacks). This poses a threat in security contexts, where adversaries may craft code snippets that evade detection by exploiting the model's structural weaknesses. Wu et al. [Wu+25] conducted a large-scale empirical study on this topic, highlighting the lack of theoretical understanding around GNN robustness and proposing a set of evaluation metrics and design principles to guide future development. Their work emphasizes that current defenses against adversarial attacks are often empirical and dataset-specific, with limited generalizability. Robustness, therefore, remains a pressing challenge for deploying GNNs in real-world security systems, especially where adversarial resilience is essential.

These strengths and limitations suggest that while GNNs are a powerful tool

for modeling program semantics, they must be applied thoughtfully—often in combination with data quality improvements, augmentation strategies, and explanation techniques—to fulfill their potential in practical vulnerability detection scenarios.

2.3 Challenges in Learning-Based Vulnerability Detection

Despite promising advances in applying machine learning to software vulnerability detection, several critical challenges continue to hinder the effectiveness and reliability of current models. These challenges arise from the inherent complexity of programming languages, the limitations of available datasets, and the opaque nature of deep learning models. In particular, issues such as data quality, class imbalance, label noise, spurious correlations, and lack of model interpretability limit both performance and trustworthiness in real-world scenarios. Addressing these problems is essential to transition from theoretical success to practical deployment in cybersecurity tools. The following subsections detail the most pressing technical and methodological obstacles facing the field.

2.3.1 Dataset Quality and Label Noise

One of the most persistent obstacles in learning-based vulnerability detection is the limited quality of available datasets. Most benchmark datasets used in this domain suffer from a combination of issues: label noise, data duplication, class imbalance, and limited diversity of examples. These deficiencies directly influence the model's ability to generalize to real-world scenarios and are often the root cause behind brittle behavior and inflated performance metrics.

Label noise remains a pervasive concern. This issue arises when examples are incorrectly annotated—e.g., benign code fragments labeled as vulnerable or vice versa. This can happen due to limitations in static analysis tools used for annotation, misunderstandings of the vulnerability context, or reliance on coarse-grained rules during dataset construction. For instance, Croft et al. (2023) reported that mislabeling rates in public datasets can range from 20% to over 70% [CBK23], severely affecting model learning and evaluation reliability. In some cases, such noise may stem from automated labeling based on commit messages, flawed use of static analysis tools, or ambiguous definitions of what constitutes a vulnerability [Din+24]. High levels of label noise can lead to models that learn incorrect associations, reducing their ability to distinguish truly vulnerable patterns from benign ones.

Duplication is another critical issue. Dataset duplication rates can range from 17% to as high as 99%, leading to overfitting and memorization of superficial patterns rather than learning generalized representations [GB23]. This also undermines model evaluation, as duplicated or near-duplicated samples may appear across both training and test splits, inflating reported performance metrics and masking the model's true generalization capacity.

Class imbalance further compounds these issues. In most real-world repositories, vulnerable functions are rare compared to benign ones. Consequently, datasets often reflect this skew, leading to imbalanced training data where models become biased toward predicting the majority class. This results in poor recall for vulnerable cases, which are typically the most critical to detect.

A related challenge is the semantic similarity and variation between vulnerable and benign examples. In some datasets, the differences between classes are minimal or subtle, such as a missing null-check or an off-by-one error. When inter-class examples (e.g., vulnerable vs. benign) are too similar, models struggle to detect the subtle distinctions that determine vulnerability. Conversely, excessive intra-class variation can confuse the learning process by introducing noise and making patterns harder to identify [Liu+22].

In response to these challenges, newer datasets have been developed to improve data quality. **PrimeVul** [Din+24] incorporates stricter deduplication protocols and manual validation to reduce label noise and improve dataset reliability. DiverseVul [Che+23] addresses semantic coverage by introducing linguistic and structural diversity across examples, enhancing the model's ability to generalize across coding styles and contexts. Despite these advances, high-quality, balanced, and diverse datasets remain scarce, especially for many individual CWE classes, posing a substantial obstacle to advancing robust vulnerability detection models.

2.3.2 Spurious Correlations and Shortcut Learning

A major challenge in machine learning-based vulnerability detection lies in the model's tendency to exploit **spurious correlations**—superficial patterns in the training data that correlate with target labels but do not reflect true causal relationships (see Figure 2.7). This phenomenon, also known as shortcut learning, results in models that appear accurate on in-distribution benchmarks but fail to generalize to new or slightly altered code contexts [Sag+20].

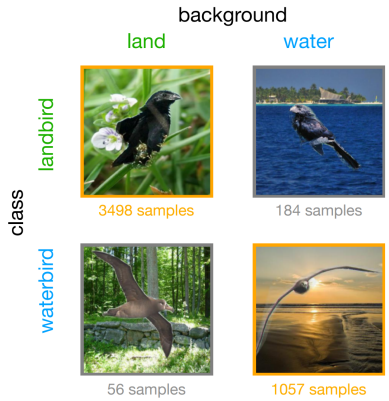


Figure 2.7: The Waterbirds dataset demonstrates spurious correlation: most waterbirds appear on water backgrounds and most landbirds on land backgrounds. A model may learn to associate the background with the bird type, rather than focusing on the bird itself. Image from [Qia+25].

In the context of source code, spurious correlations often emerge from biases in the dataset construction process. For instance, duplicated vulnerable samples with specific variable names, function signatures, or even formatting styles can lead models to associate those features with vulnerabilities. Rather than learning semantic indicators of faulty logic or missing validations, the model learns to recognize stylistic artifacts that are coincidentally frequent among vulnerable examples. This behavior undermines the core goal of vulnerability detection—understanding the underlying program behavior that leads to insecure execution.

Figure 2.8 highlights an example of spurious correlation through two semantically similar functions. In the upper (benign) function, the variable `mode` is safely assigned from an internal configuration source (`aa_g_profile_mode`). In contrast, the lower (vulnerable) function uses a user-provided `mode` value directly in a `sprintf()` call without proper bounds checking. A model trained predominantly on benign patterns like the former may incorrectly associate the presence of `mode` with safe behavior, overlooking its insecure usage when derived from untrusted input.

Original benign code: No CWE-20 issue.

```
1 static int net_get_rate(struct wif *wi)
2 {
3     struct priv_net *pn = wi_priv(wi);
4
5     return net_cmd(pn, NET_GET_RATE, NULL, 0);
6 }
```

Vulnerable counterexample: Unvalidated `user_input` introducing a CWE-20 flaw

```
1 static int net_get_rate(struct wif *wi, int user_input)
2 {
3     struct priv_net *pn = wi_priv(wi);
4
5     // Introduced vulnerability: accepting user input without validation
6     return net_cmd(pn, NET_GET_RATE, &user_input, sizeof(user_input));
7 }
```

Figure 2.8: Illustration of spurious correlation in source code. The upper (benign) function assigns a trusted internal value to `mode`, while the lower (vulnerable) function uses unchecked `user_ input`. Without the counterexample, a model may wrongly associate the variable `mode` with safe behavior overlooking its unsafe usage.

Such risks are especially pronounced in scenarios involving minimal differences between classes. For example, a vulnerable function and its benign variant might differ only by a single null-check or input sanitization condition—like the example depicted in Figure 2.8. A model trained on biased data might still label both as vulnerable if it has learned to overfit on tokens like function names or constants. This behavior not only produces false positives but also damages trust in the system's reliability.

Recent studies on machine learning spuriousness [Ye+24; BW24; Ste+24] have underscored how widespread this issue is, even in high-performing models. These studies argue that statistical correlations learned from data are not sufficient for robust decision-making, particularly when spurious features are easier to learn than the underlying semantics. In vulnerability detection, this is especially dangerous, as

overfitting to syntactic patterns can render models brittle and unfit for real-world deployment.

Addressing this issue requires both improved data practices and more principled model training. Strategies that enrich the semantic content of training data are gaining attention for encouraging models to focus on true vulnerability semantics. Similarly, training-time interventions (e.g., loss regularization or adversarial training) can encourage the model to learn more robust features. However, detecting and quantifying spuriousness remains a difficult task, and most existing approaches lack a rigorous framework to identify when models are exploiting shortcuts.

2.3.3 Data Augmentation Strategies for Source Code

Data augmentation has become an increasingly important strategy in machine learning for addressing data scarcity, improving generalization, and mitigating class imbalance. In the context of source code vulnerability detection, augmentation techniques aim to produce new training examples while preserving both the syntactic validity and semantic meaning of the original functions. This requirement is particularly challenging in source code settings, where minor modifications can introduce compilation errors or unintentionally change program behavior.

Several augmentation methods have been recently proposed to enhance learning-based vulnerability detection. CodeGraphSMOTE [Gan+23] is one prominent strategy which adapts the classical SMOTE technique to structured graph representations. It performs interpolation in the latent embedding space of graph neural networks (GNNs), effectively synthesizing new samples that lie between existing vulnerable and benign examples. While this method balances the dataset, it does so at a high level of abstraction, making it harder to interpret or control the semantic fidelity of the resulting samples.

Another family of approaches focuses on transformation-based augmentation, which modifies the source code directly through label-preserving transformations. These include renaming identifiers, altering loop structures, reordering independent statements, and substituting equivalent logic constructs. Such transformations aim to increase the structural diversity of the training data while maintaining the original labels [Liu+24]. However, their effectiveness depends on the coverage and diversity of transformation templates, and they do not create challenging contrastive examples that could push the model to learn finer-grained semantics.

More recently, LLM-based code synthesis has emerged as a powerful augmentation tool. For example, VulScribeR [Dan+24] leverages large language models (LLMs) to generate synthetic vulnerable code samples based on real-world vulnerability patterns. These approaches benefit from the generative capacity of LLMs and can produce diverse examples at scale. Nonetheless, they typically rely on predefined vulnerability prompts and lack guarantees about semantic contrast or proximity to real-world decision boundaries. The Figure 2.9 depicts the overview of the VulScribeR Augmentation approach.

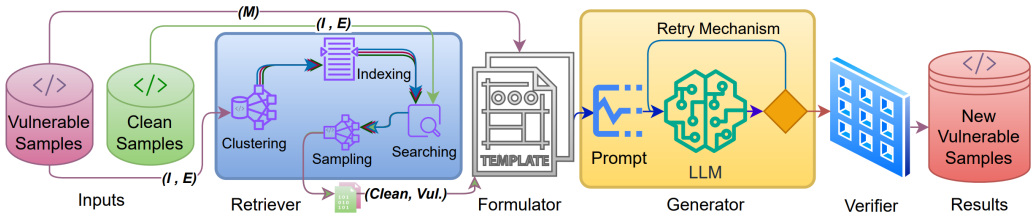


Figure 2.9: Overview of the VulScribeR augmentation system [Dan+24]. The architecture includes four key components: a Retriever (used by the Injection and Extension strategies) to fetch example pairs, a Formulator that instantiates prompt templates for each strategy (Mutation, Injection, or Extension), a Generator that uses an LLM to produce candidate vulnerable samples, and a Verifier that filters outputs using a fuzzy C parser (Joern) to eliminate syntactic errors. The system balances diversity and realism, providing scalable LLM-based augmentation for vulnerability datasets.

In contrast to the above, counterfactual data augmentation remains relatively underexplored in source code analysis. This technique involves making minimal and targeted edits to existing functions such that the vulnerability label flips—e.g., by removing an input validation check or introducing a tainted data flow. Counterfactuals serve as semantically meaningful near-neighbors on opposite sides of the decision boundary, providing stronger learning signals for classification models. Importantly, they allow researchers to test a model's sensitivity to small but security-relevant changes and help mitigate shortcut learning. Despite its promise, few works have systematically incorporated counterfactual examples in training pipelines for vulnerability detection, leaving this as an open area of research.

2.3.4 Lack of Model Interpretability

Interpretability remains a persistent barrier to the adoption of machine learning systems for vulnerability detection. While models such as Graph Neural Networks (GNNs) and Transformers can achieve high performance, they often operate as black boxes—offering predictions without insight into the reasoning behind them. In a critical domain such as cybersecurity, this opacity undermines trust, limits feedback, and complicates debugging, especially when misclassifications occur.

To address this, several explanation methods have been developed. Model-agnostic techniques like LIME [RSG16] and SHAP [LL17] are widely used to estimate feature contributions by perturbing inputs and analyzing changes in output. These tools have proven valuable in many domains, but their granularity and abstraction level are often too coarse for reasoning over structured code graphs or semantics.

Definition of Counterfactuals

In machine learning, a counterfactual refers to a minimally modified version of an input instance that results in a different model prediction. Counterfactuals are designed to answer "what-if" questions by identifying the smallest changes

necessary to flip a model's output—thereby revealing which aspects of the input are most influential in the decision-making process [Ver+22].

Formally, for a model $f(\cdot)$, given an input x with prediction $f(x) = y$, a counterfactual x' satisfies $f(x') = y'$ where $y' \neq y$, and the difference $\Delta(x, x')$ is minimal under some task-relevant similarity constraint. In the context of source code, a counterfactual could involve inserting a missing input check, modifying a control condition, or altering a data flow in a way that changes the vulnerability label while maintaining syntactic validity.

Counterfactuals are particularly valuable in explainability, as they provide concrete examples of how and why decisions change. They are also increasingly used to improve model robustness by exposing models to decision-boundary-adjacent examples during training, helping to prevent reliance on spurious or non-generalizable features [WMR18].

Explainability in Source Code and Graph-Based Models

In code analysis, domain-specific solutions have emerged. IVDetect [LWN21] employs program dependency graphs to trace vulnerabilities back to influential operations and data flows. Similarly, CFExplainer [Chu+24] applies counterfactual reasoning, generating minimally modified inputs to expose decision boundaries. These approaches bring greater semantic clarity to explanations but still face challenges in scalability and applicability across architectures. Figure 2.10 depicts the factual reasoning and what-if analysis capabilities (top) of CFExplainer, along with a detailed architectural overview of its core components (bottom).
For graph-based models like GNNs, dedicated explanation techniques have been introduced. GNNExplainer[Yin+19] highlights relevant nodes and edges that influence predictions by optimizing for mutual information between subgraphs and output classes. PGM-Explainer[VT20] goes a step further by modeling causal relationships in predictions using probabilistic graphical models. These methods yield interpretable substructures but often involve computationally expensive post hoc optimization and can be unstable across different inputs. Figure 2.11 illustrates the architecture of PGM-Explainer, which explains GNN predictions through probabilistic graphical models (PGMs) by combining data generation, variable selection, and structure learning.

More recently, **Illuminati** [HJH23] was introduced as a domain-specific GNN explainer tailored for cybersecurity tasks. It identifies the most influential nodes, edges, and attributes in a prediction and produces minimal, sufficient subgraphs that clarify the model's internal decision logic. Illuminati combines scalability with precision and is well-suited for structured code analysis. However, it operates purely in a post hoc fashion—revealing the reasoning behind decisions without influencing or improving model training. Furthermore, like most explainers, it does not directly address the issue of spurious correlations or shortcut learning, which can corrupt attribution quality even when visualizations are provided.

Despite these advances, interpretability methods still fall short of delivering explanations that are both faithful to the model and actionable for practitioners.

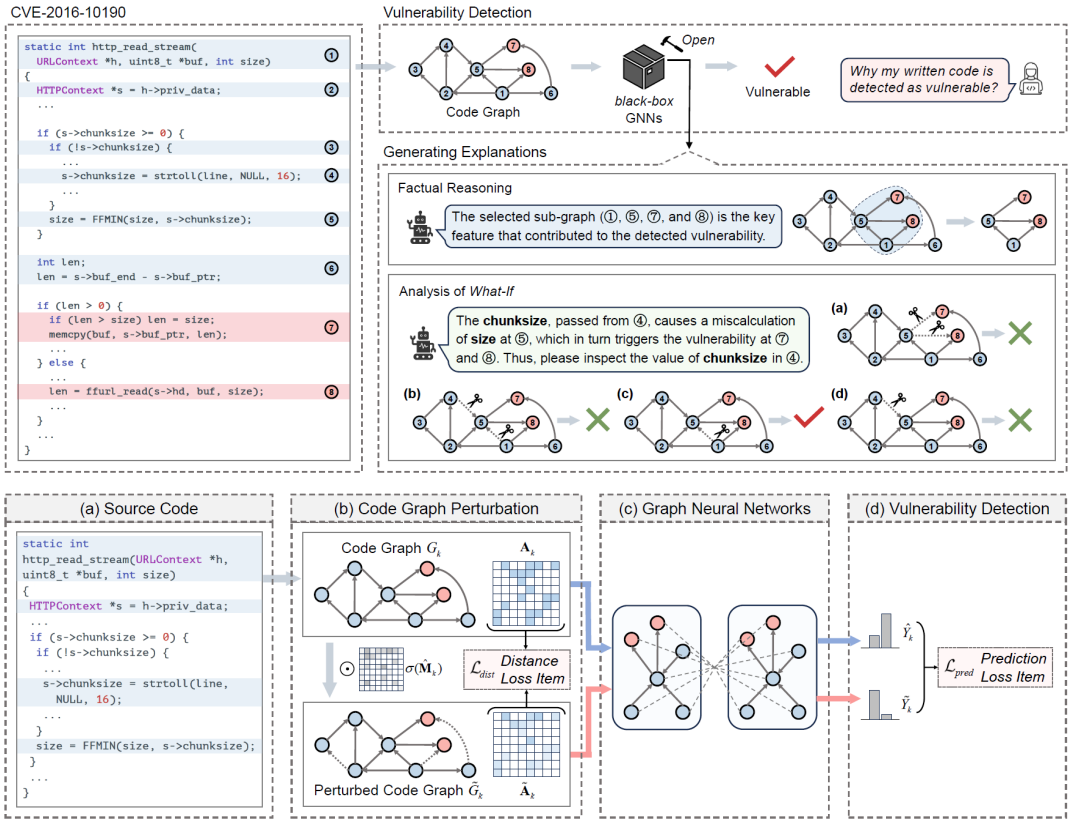


Figure 2.10: CFExplainer, a counterfactual reasoning-based explainer for GNN-based vulnerability detection. Top: Comparison between factual reasoning and counterfactual what-if analysis, illustrating how CFExplainer identifies minimal perturbations that alter predictions. Bottom: Architectural overview showing the three main components—code graph perturbation via edge masks, counterfactual reasoning framework, and explanation generation. Adapted from the original paper [Chu+24].

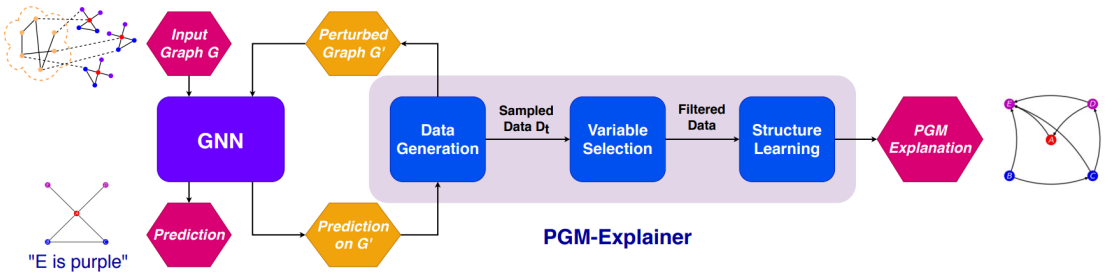


Figure 2.11: Architecture of PGM-Explainer [VT20], a probabilistic graphical model-based explainer for GNNs. The framework includes three main steps: (1) data generation through graph perturbations and prediction logging, (2) variable selection to filter out irrelevant features, and (3) structure learning to build a compact PGM that captures the dependencies underlying the prediction. Adapted from the original paper.

In the context of software security, effective explanation tools must illuminate semantically meaningful code patterns, align with developer expectations, and

expose vulnerabilities in logic—not just artifacts in the dataset. Bridging this gap remains a critical direction for future research.

2.3.5 Generalization and Robustness

Ensuring that vulnerability detection models generalize effectively to unseen or adversarially altered code is crucial for real-world deployment. Generalization extends beyond fitting training data—it encompasses handling code variations, diverse coding styles, platform-specific idioms, and maliciously crafted perturbations. Incomplete representation of these factors in the training data can result in models that overfit to superficial patterns, suffering dramatic performance drops on real-world or adversarial examples.

Recent studies in graph learning have highlighted this vulnerability demonstrate that GNNs trained on highly regular graphs lack diversity and exhibit poor generalization across structural patterns. Moreover, [Wu+25] a large-scale empirical analysis was conducted across architectures and datasets, finding that GNN robustness hinges critically on graph-structure diversity and model capacity; notably, adversarial perturbations exhibit asymmetric transferability between models.

Adversarial attacks against GNNs come in various forms: feature perturbations, insertion of malicious nodes or edges ("injection attacks"), and subtle graph modifications ("modification attacks"). These can severely degrade model performance, even when perturbations are imperceptible. Defense mechanisms have emerged—such as GNNGuard [ZZ20], which prunes suspicious edges based on feature similarity, and Hamiltonian Neural Flow architectures that enforce stability constraints to resist topological manipulations.

Recent work exploring structural sparsification and coarsening has shown mixed effects: sparsification can improve robustness against certain attacks, though coarsening may amplify vulnerability to others. Certification-based approaches, like GCORN [Abb+24], link model robustness to orthonormal constraints on weight matrices, offering provable guarantees against node feature attacks. These strategies highlight a shift towards combining empirical defense with formal robustness measures.

In the context of vulnerability detection, such generalization needs translate to resilience against code variants, injection of malicious patterns, or deletion/editing of validation logic. Ensuring that models do not rely on dataset artifacts but genuinely learn semantic causal features is critical. Integrating adversarial training, structural regularization, and certification methods—as seen in robust GNN research—can enhance real-world viability.

Chapter 3

VISION Framework

To address the limitations observed in current vulnerability detection models—such as lack of robustness, susceptibility to spurious correlations, and poor interpretability—this work proposes a unified framework named VISION (Vulnerability Identification and Spuriousness Mitigation via Counterfactual Augmentation). VISION is designed to improve the reliability and transparency of Graph Neural Network-based detectors by systematically incorporating counterfactual data during training and enabling post hoc graph-based explainability. The framework integrates state-of-the-art tools for code graph generation, explanation, and visualization, and is empirically validated on a large-scale dataset focused on CWE-20 (Improper Input Validation) vulnerabilities. This chapter provides a detailed description of the core components of the framework, from data preparation and augmentation to model training and interpretability modules.

3.1 Overview of the VISION Pipeline

This section introduces the overall architecture of the proposed VISION framework (Vulnerability Identification and Spuriousness Mitigation via Counterfactual Augmentation), which is designed to enhance both the robustness and interpretability of machine learning-based systems for software vulnerability detection. The framework is motivated by key shortcomings in current approaches, such as their overreliance on superficial patterns, poor generalization to real-world code, and the limited transparency of Graph Neural Network (GNN) predictions. To overcome these challenges, VISION systematically augments training data with counterfactual examples and embeds explainability mechanisms into its architecture.

The pipeline begins with the **PrimeVul** dataset [Din+24], a recent large-scale and high-quality resource for source code vulnerability detection. From this dataset, only instances labeled under the **CWE-20 vulnerability** category—Improper Input Validation [MIT25]—are extracted. This class is particularly relevant due to its real-world significance, well-defined semantics, and availability of thousands of instances. After filtering, the resulting subset is used as the foundation for constructing a balanced and semantically controlled training corpus.

To improve model generalization and reduce reliance on dataset artifacts, VISION

incorporates **counterfactual data augmentation**. These counterfactuals are defined as minimally modified versions of code samples that preserve syntactic correctness while reversing their vulnerability label. For instance, a safe function can be turned vulnerable by removing an input validation check, and vice versa. This augmentation is achieved through prompt-based interaction with a Large Language Model (LLM), specifically GPT-4o-mini, which rewrites functions according to dynamically generated instructions. After generation, each pair of original and counterfactual samples is rigorously filtered and validated to ensure label consistency and code integrity. The final curated set forms the ***CWE-20-CFA benchmark***, composed of both original and counterfactual code samples balanced across classes.

Once the dataset is prepared, each function is parsed using Joern [Whi25], a static analysis tool that constructs Code Property Graphs (CPGs) by combining structural representations such as abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs) [Yam+14]. The resulting graphs capture semantic and syntactic properties of code and serve as the input format for the GNN model.

For vulnerability prediction, VISION builds on the Devign architecture [Zho+19], a GNN specifically tailored for analyzing code graphs. Devign employs gated graph recurrent layers to iteratively propagate information across the graph, followed by convolutional layers for classification. Each function, now represented as a graph with node and edge embeddings, is passed through the model to predict its vulnerability status. Training is conducted on datasets composed of paired original and counterfactual examples, allowing the model to learn subtle differences that truly define vulnerabilities rather than spurious cues.

Beyond prediction, interpretability plays a central role in the VISION framework. To this end, the model is integrated with **Illuminati** [HJH23], a domain-specific explainer for GNNs in cybersecurity. Illuminati identifies the most influential subgraphs responsible for a model's prediction, providing insight into the decision-making process. These subgraph-based attributions allow for a detailed understanding of how the model arrives at a particular classification and help verify whether predictions are based on semantically meaningful evidence.

Finally, VISION incorporates an interactive **visualization module** that enables human-in-the-loop analysis. This interface highlights both code and graph elements using color-coded importance scores derived from the Illuminati explainer. Users can interactively explore which code statements were most influential, how the model's attention shifts across different training settings, and whether key vulnerability-inducing components were correctly identified. This module is particularly valuable for debugging, auditing model behavior, and building trust in real-world security applications.

A full depiction of the VISION architecture is shown in Figure 3.1, illustrating the end-to-end flow from raw data to prediction and explanation.

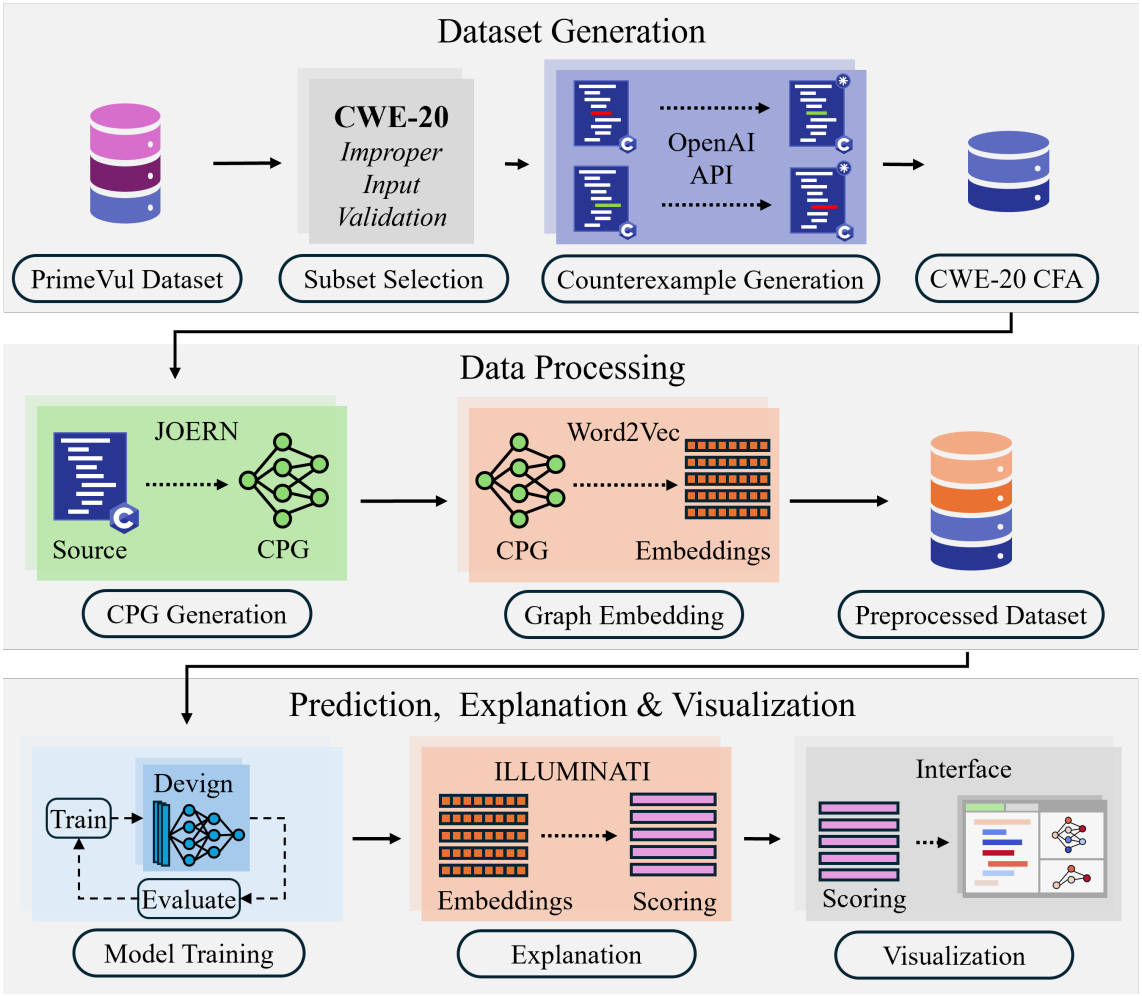


Figure 3.1: Beginning with the original PrimeVul dataset, the framework outlines a complete pipeline: filtering for CWE-20 samples, generating counterfactuals for class balancing, constructing graphs using Joern-generated CPGs, extracting embeddings, training the Devign model, and ultimately generating explanations with Illuminati, complemented by a visualization module for interpretability.

3.2 Dataset Selection and Preparation: CWE-20 Vulnerability

A fundamental component of the VISION framework is the use of a high-quality, semantically diverse, and well-labeled dataset to ensure the effectiveness of the proposed training and evaluation strategy. To this end, the dataset used in this work is derived from PrimeVul [Din+24], a recent benchmark collection specifically curated for training vulnerability detection models on source code. PrimeVul provides a robust foundation for this research due to its combination of real-world code examples, reduced label noise, and structurally diverse vulnerability patterns.

3.2.1 PrimeVul Dataset for Vulnerability Detection

The PrimeVul dataset was designed to overcome many of the known limitations in earlier vulnerability detection benchmarks, such as duplicated code, mislabeled

samples, and poor structural variation. These deficiencies are widely documented in the literature and are known to contribute to overfitting, spurious correlations, and unreliable generalization [CBK23; GB23]. PrimeVul addresses these issues by applying rigorous de-duplication techniques, human-in-the-loop label verification, and stricter filtering rules that aim to retain only well-formed and semantically meaningful code samples.

In total, PrimeVul contains over 224,000 functions extracted from real-world, open-source C and C++ projects, each labeled for one or more Common Weakness Enumeration (CWE) classes. For the scope of this work, the dataset is filtered to include only those examples corresponding to CWE-20: Improper Input Validation [MIT25]. This vulnerability type, consistently ranked among the most critical software weaknesses by organizations like MITRE and NIST [MIT22], is particularly suitable for experimentation due to its semantic clarity, practical relevance, and availability within PrimeVul. An overview of this filtering process is illustrated in Figure 3.2.

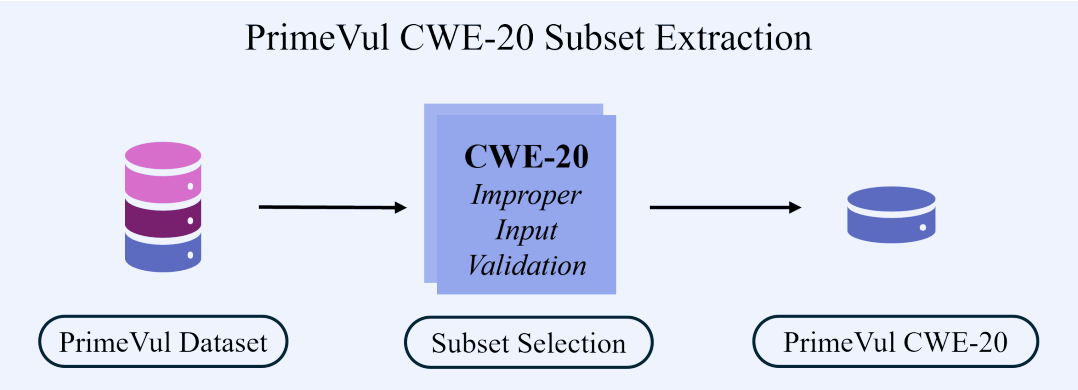


Figure 3.2: Overview of the CWE-20 dataset filtering process. Starting from the complete PrimeVul dataset, a subset selection stage filters only the samples labeled with CWE-20 (Improper Input Validation). This targeted extraction produces the focused training corpus used throughout the VISION framework, balancing semantic clarity, real-world relevance, and data availability for counterfactual generation and GNN training.

3.2.2 Motivation for Focusing on CWE-20

The decision to focus exclusively on CWE-20 vulnerabilities in this research is driven by three main considerations:

- **Clarity of Semantics**: CWE-20 vulnerabilities are characterized by the absence or failure of input validation mechanisms—such as bounds checking, null verification, or type sanitization. These issues frequently manifest through concrete, well-understood patterns, including unchecked user inputs passed to sensitive APIs. This clarity makes CWE-20 particularly appropriate for tasks such as counterfactual generation, where meaningful, minimal modifications must invert the vulnerability label.

- **Data Availability**: The PrimeVul dataset includes approximately 15k CWE-20 samples. These comprise both benign (non-vulnerable) and vulnerable examples, with sufficient quantity and structural variation to support model training, statistical evaluation, and counterfactual pairing. Importantly, this volume of data permits the construction of balanced training sets without excessive upsampling or synthetic over-generation, ensuring more realistic learning conditions.
- **Real-World Relevance**: Improper Input Validation is one of the most commonly exploited classes of vulnerabilities, as evidenced by its inclusion in the CWE Top 25 Most Dangerous Software Weaknesses [MIT22]. Real-world security incidents frequently involve input vectors that are malformed, unchecked, or manipulated, making this vulnerability class highly impactful for applied machine learning-based detection systems.

CWE-20 in Practice: An Illustrative Example

To concretely illustrate the kind of issues CWE-20 encompasses, consider the code snippet shown in Figure 3.3. The function `validGlxScreen` takes an integer index screen and returns a pointer to a corresponding screen resource. In the current implementation, it checks whether the screen index is greater than or equal to the number of available screens but fails to verify whether the index is negative. This omission creates an opportunity for invalid memory access, as a negative index could result in an illegal array dereference or pointer arithmetic error.

```
1  validGlxScreen(ClientPtr client, int screen, __GLXscreen **pGlxScreen, int *err) {
2      if (screen >= screenInfo.numScreens) {
3          client->errorValue = screen;
4          *err = BadValue;
5          return FALSE;
6      }
7      *pGlxScreen = glxGetScreen(screenInfo.screens[screen]);
8
9      return TRUE;
10 }
```

Figure 3.3: The `validGlxScreen` function ensures that the `screen` index does not exceed the number of available screens but fails to check for negative values. This can result in invalid array access and illustrates an improper input validation flaw.

3.2.3 Filtering and Preprocessing

To prepare the dataset for integration with the VISION pipeline, a focused subset of the PrimeVul data is extracted. The filtering process involves:

- Selecting only functions labeled with CWE-20.
- Removing duplicate entries based on function hash and structural similarity.
- Standardizing the format of code samples to ensure compatibility with downstream tools such as Joern.
- Retaining only complete functions with well-formed syntax.

The resulting subset comprises 14,944 CWE-20-labeled samples, divided into approximately 14,473 benign examples and 471 vulnerable ones (see Table 3.1). This stark imbalance reflects the naturally skewed distribution of vulnerabilities in software but poses a challenge for training robust models. To address this, the next section describes how this base dataset is augmented through the generation of synthetic counterfactuals to construct a fully balanced training corpus referred to as CWE-20-CFA.

Table 3.1: PrimeVul Dataset CWE-20 Vulnerability Filtering

Dataset Stage	Benign	Vulnerable	Total
PrimeVul	218,529	6,004	224,533
CWE-20 Filtered	14,473	471	14,944

3.3 Counterfactual Generation and Dataset Augmentation

To reduce dataset bias and spurious correlations, this work introduces a counterfactual data augmentation strategy that helps the model focus on truly security-relevant signals. The approach involves generating minimally different function pairs with opposite vulnerability labels, encouraging the model to learn subtle semantic distinctions between secure and insecure code.

This counterfactual generation process is summarized visually in Figure 3.4. Each function in the CWE-20 subset is passed through the LLM, which outputs a semantically aligned version with the opposite label. By pairing original and counterfactual functions and consolidating them into a balanced dataset, the CWE-20-CFA corpus enhances the model's ability to learn from meaningful variations, rather than relying on superficial cues or dataset artifacts.

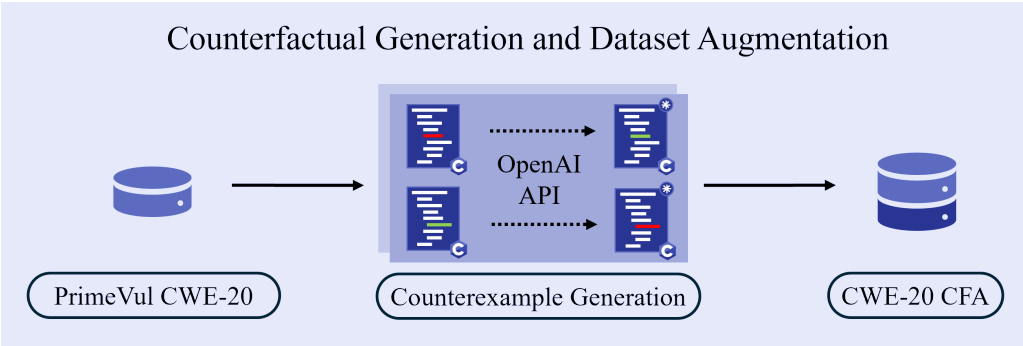


Figure 3.4: Overview of the counterfactual generation and dataset augmentation process. Starting from the CWE-20 subset of PrimeVul, original code samples are sent to the OpenAI API, which generates minimally modified counterparts with flipped vulnerability labels. The resulting original–counterfactual pairs form the CWE-20-CFA dataset, used to train models that are more robust to spurious correlations and better equipped to learn decision-relevant patterns.

3.3.1 Formal Definition of Code Counterfactuals

In the context of vulnerability detection, a counterfactual is defined as a minimally edited version of a code function such that its vulnerability label is flipped—transforming a benign function into a vulnerable one or vice versa—while preserving its syntactic and semantic integrity. The goal is to ensure that both the original and counterfactual remain valid, compilable, and realistic representations of functional source code.

This definition draws from counterfactual explanations in machine learning [WMR18], where explanations take the form of *nearest neighbors* on the opposite side of a decision boundary. The concept also aligns with contrastive learning frameworks [KHL20; RMP21], where the presence of fine-grained semantic differences is critical for robust learning.

Formally, let $f(\cdot)$ be a trained vulnerability classifier operating on code graphs. Given a graph $G = (V, E)$, with node set $V = \{v_1, \ldots, v_N\}$ and edge set $E = \{(v_i, v_j)\}$, suppose $f(G) = y$ and $f(G') = y_c$ such that $y \neq y_c$. A counterfactual mapping $e : G \to G'$ is one that introduces a minimal change in the structure or content of G such that the model's output is flipped.

Counterfactuals offer two key benefits for improving the effectiveness of vulnerability detection systems.

- Counterfactuals enable **dataset balancing** by generating semantically paired examples for both vulnerable and benign classes. For every original function, a corresponding counterexample is created by introducing or removing a vulnerability with minimal semantic modification. This ensures a more even class distribution, which is particularly valuable in domains like software security where vulnerable samples are often scarce.
- Counterfactuals introduce **fine-grained**, **security-relevant changes** that help models learn subtle distinctions associated with real vulnerabilities. By exposing the model to function pairs that differ only in critical vulnerability-inducing elements, counterfactuals force it to focus on meaningful patterns rather than superficial cues—improving generalization.

In contrast to traditional augmentation techniques, such as token shuffling or generic code transformations, counterfactuals preserve the original structure and style of real-world code while introducing deliberate, interpretable shifts at the decision boundary. This maintains the fidelity of the training data and enhances the model's ability to detect nuanced vulnerabilities in practice.

Unlike traditional augmentation techniques—such as token shuffling or code transformations—counterfactuals retain fidelity to natural programming styles while introducing meaningful decision-boundary shifts.

An example of such a counterfactual transformation is shown in Figure 3.5. The upper function represents a benign implementation in which the variable `mode` is securely assigned from an internal configuration parameter before being passed to

`sprintf()`. In contrast, the counterfactual version accepts `mode` as an external input and omits validation checks, introducing a CWE-20 vulnerability. This minimal but semantically impactful modification highlights the essence of counterfactuals in code: maintaining high structural similarity while altering security semantics to help the model learn truly discriminative features.

Original benign code: No CWE-20 issue.

```
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]);
}
```

Vulnerable counterexample: Unvalidated `mode` input introducing a CWE-20 flaw

```
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 the bounds of mode
    return sprintf(buffer, "%s", profile_mode_names[mode]);
}
```

Figure 3.5: Illustration of a counterfactual code pair. The upper function is the original benign version, where the `mode` variable is safely assigned from an internal source. The lower function represents the counterfactual vulnerable variant, in which `mode` is passed as an unvalidated external input to `sprintf()`, introducing a CWE-20 (Improper Input Validation) flaw.

3.3.2 Counterfactual Generation Strategy

To create these counterfactuals at scale, a prompt-based rewriting strategy is adopted using GPT-4o-mini via the OpenAI API. The approach consists of generating a prompt that describes the vulnerability context and instructs the model to rewrite a function to either introduce or remove a CWE-20 vulnerability. This process is fully automated and allows the framework to create semantically controlled, label-inverting examples with minimal human supervision.

The prompt is dynamically assembled for each sample. If the original function is benign, the instruction asks the model to inject a CWE-20 vulnerability by removing a validation check or introducing unsafe input usage. If the function is already vulnerable, the instruction asks for a secure version of the same logic by adding proper validation mechanisms.

Each generated counterfactual is subject to post-processing and filtering. This includes:

- Syntax validation using a C parser (e.g., via Joern).
- Deduplication.

- Manual spot checks on a small subset to verify semantic coherence.

This results in the **CWE-20-CFA dataset**, composed of balanced pairs of original and counterfactual examples. These examples are used to train the Devign model in a way that emphasizes decision-critical variations rather than spurious correlations.

Prompt Template

To generate high-quality counterfactuals, this project employs a prompt-based rewriting strategy leveraging a lightweight LLM (GPT-4o-mini) via the **OpenAI API**. Each counterfactual is created by sending the original source code and its label as input to a dynamically crafted prompt.

```python
def generate_counterexample(example: pd.Series) -> pd.Series:
    """
    Use OpenAI API to generate a modified version of input_code.
    If the target_label is 1 (vulnerable), request a benign version.
    If the target_label is 0 (benign), introduce a vulnerability.
    """
    cwe = example.cwe[0] # First Vulnerability (e.g. CWE-20)

    prompt_template = f"""
    The following is a {'vulnerable' if example.target == 1 \
    else 'benign'} C function.

    Please {'remove the vulnerability to make it safe' if example.target == 1 \
    else f'introduce a security vulnerability (cwe: {cwe})'}.

    Original function:
    ```c
    {example.func}
    ```

    Modified function:
    """

    try:
        response = openai_client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt_template}],
            stream=False,
            temperature=0.7
)
        # Get response content
        response_content = response.choices[0].message.content
        # Extract function code
        if "```" in response_content:
            ce_func = response_content.split("```")[1].strip()

            if ce_func.startswith("c\n"):
                ce_func = ce_func[2:]
        # Create pandas series
        ce = pd.Series(data=[ce_func, 0 if example.target else 1, cwe, \
            example.func], index=["func", "target", "cwe", "orig_func"])

        return ce

    except Exception as e:
        print("Error:", e)

        return None
```

Figure 3.6: Python function for constructing the prompt template used in counterfactual generation. Depending on the original label, the function dynamically instructs the LLM to either introduce or eliminate a vulnerability in the C function, explicitly referencing its CWE class.

The prompt instructs the LLM to either introduce or eliminate a vulnerability depending on the original label. Specifically, if the original sample is labeled as vulnerable, the model is asked to modify it into a safe (benign) variant. Conversely, for benign samples, the prompt requests the injection of a vulnerability matching the original CWE class—in this case, CWE-20.

This template ensures both syntactic realism and semantic relevance in the generated code. It maintains a natural-language formulation while using clear delimiters (e.g., triple backticks for code blocks) to preserve formatting. The entire counterfactual generation process is automated using a function like the one shown in Figure 3.6, which builds the prompt string based on the input function and its associated label. This flexible format allows tailoring prompts to any CWE class and can be extended to other vulnerability types in future work.

Once the prompt is constructed, it is sent to the OpenAI API using the GPT-4o-mini model, with a temperature parameter set to 0.7 to allow for controlled variability in the generated outputs. The model's response is parsed to extract the modified function code, ensuring that it is properly delimited and stripped of formatting artifacts. If the transformation is successful, a new `pandas.Series` is returned containing the counterfactual function, its updated label, CWE class, and a reference to the original code. Error handling is also integrated to catch and report any issues with API calls or malformed responses, ensuring robustness and reliability during large-scale generation.

3.3.3 CWE-20-CFA: Balanced Augmented Dataset

The result of the filtering, augmentation, and validation process is the construction of the CWE-20-CFA benchmark—a balanced, semantically controlled dataset composed of original and counterfactual examples. Each class contains an equal number of functions, and both sides are symmetrically structured to capture meaningful differences in vulnerability semantics. Table 3.2 summarizes the dataset composition across key stages of transformation.

Table 3.2: Final statistics of the CWE-20-CFA dataset used in this work. Original functions from the PrimeVul CWE-20 subset were augmented with validated counterfactuals to ensure balance and structural alignment across classes.

Dataset Stage	**Benign**	**Vulnerable**	**Total**
PrimeVul (All CWEs	218,529	6,004	224,533
CWE-20 Filtered Subset	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

3.3.4 Preprocessing: Code Graph Construction and Embedding

To transform raw source code into structured inputs for GNN-based vulnerability detection, this work implements a multi-stage preprocessing pipeline. The process involves generating Code Property Graphs (CPGs) from source code using the Joern framework, followed by graph parsing and embedding using a Word2Vec-based encoder to produce node feature representations and graph connectivity suitable for model input. The full preprocessing pipeline is illustrated in Figure 3.7, showing the transformation from raw source code to fully embedded graph inputs for model training.

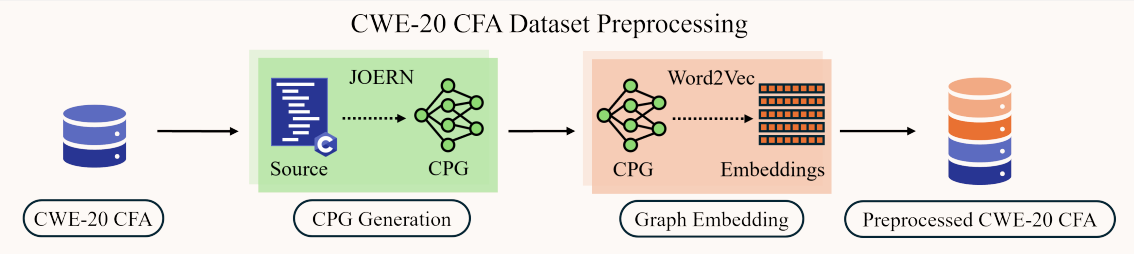


Figure 3.7: Overview of the preprocessing pipeline for the CWE-20-CFA dataset. The process begins with raw source code from the counterfactually augmented dataset, which is parsed by Joern to generate Code Property Graphs (CPGs). These graphs are then embedded using Word2Vec-trained token vectors to produce node-level representations. The resulting graphs, encoded with semantic and structural features, form the preprocessed dataset ready for GNN-based vulnerability classification.

Joern-Based CPG Extraction

Joern [Whi25] is a static analysis tool that constructs CPGs by combining multiple program representations, including abstract syntax trees (AST), control flow graphs (CFG), and data dependency graphs (DDG) into a unified graph format. Each counterfactual or original code function is saved as a C source file and parsed into a binary CPG using `joern-parse`. The CPG is then processed by a custom Joern script, `graph-for-funcs.sc`, that extracts function-level subgraphs and serializes them as JSON files. This modular interface allows automated parsing, conversion, and cleanup for large-scale datasets with retry logic to handle occasional parsing failures.

Graph Parsing and Node Filtering

Once extracted, the CPG JSON files are processed to extract relevant graph structures. The parser filters out irrelevant or noisy nodes (e.g., comments, unknown types), retaining those with valid code, line numbers, and semantic labels. The remaining nodes are ordered by their source line and column numbers to preserve execution structure. This process ensures consistency across training examples and supports graph neural models that rely on relative node positioning. The core logic for generating CPGs from source functions using Joern is summarized in Figure 3.8.

```
...
# Subtask 1: Code Parsing
source_file_path = os.path.join(PATHS["source"], f"{index}.c")
with open(source_file_path, 'w') as f:
    f.write(example.func)

# Parsing function to .bin
cpg_file = joern_parse(JOERN_CLI_DIR, source_file_path, PATHS['cpg'], \
                f"{index}_cpg")

# Subtask 2: Create CPG graphs JSON file
json_file = joern_create(JOERN_CLI_DIR, PATHS['cpg'], PATHS['cpg'], cpg_file)

# Subtask 3: Get CPG object from JSON
graphs = json_process(PATHS['cpg'], json_file)
cpg = graphs[0][1]  # Extract CPG
example["cpg"] = cpg
example.to_pickle(os.path.join(PATHS['cpg'], f"{index}_cpg.pkl"))

# Remove unused files
os.remove(os.path.join(PATHS['cpg'],f"{index}_cpg.bin"))
os.remove(os.path.join(PATHS['cpg'],f"{index}_cpg.json"))
os.remove(os.path.join(PATHS['cpg'],f"{index}_cpg.pkl"))
os.remove(os.path.join(PATHS['source'],f"{index}.c"))

# Add example to the dataset
dataset = pd.concat([dataset, example.to_frame().T])
...
```

Figure 3.8: Code snippet for generating and processing Code Property Graphs (CPGs) using Joern. Each function is saved as a source file, parsed into a binary graph, exported to JSON, and stored before cleanup (Full code in Figure C3).

Tokenization and Word2Vec Embedding

To embed each node into a fixed-size feature vector, the code associated with each node is tokenized using a custom tokenizer that removes literals, comments, and special characters. Tokens are mapped to vectors using a Word2Vec model trained on the entire dataset's code corpus. Each node's embedding is constructed by averaging its token vectors and concatenating them with the node type to capture both syntax and semantics. Tokens not found in the vocabulary are assigned zero vectors. The model also stores a mapping from each node ID to its original source code tokens and vectorized representation for later inspection. The embedding loop that processes CPGs into fully structured graph inputs for model training is shown in Figure 3.9.

Edge Construction and Graph Assembly

Graph connectivity is derived using only specific edge types (typically AST or DFG) to focus the model on syntactic or semantic structure. Each node's local connections are indexed and stored in a COO-style adjacency format. The resulting structure includes the edge index, node features (x), and target label (y), packed into a `torch_geometric.data.Data` object compatible with PyTorch Geometric.

This pipeline ensures that each function—original or counterfactual—is represented as a fully encoded input graph with rich semantic and structural information, enabling effective learning of subtle vulnerability patterns. Intermediate artifacts, such as CPGs and embeddings, are cached periodically for reusability and robustness against processing errors.

```
...
for index, row_series in dataset_df.copy().iterrows():

    row_df = row_series.to_frame().T

    # Function Tokenization
    tokenized_func_df = tokenize(row_df)
    func_tokens = tokenized_func_df.tokens

    # Build and Train Word2Vec Model
    w2vmodel.build_vocab(corpus_iterable=func_tokens, update=not w2v_init)
    w2vmodel.train(func_tokens, total_examples=w2vmodel.corpus_count, epochs=1)

    # Embed CPG to node representation and convert to graph
    row_df[["nodes", "nodes_by_line_map"]] = row_df.apply( \
        process_cpg_to_nodes_row, axis=1)
    row_df = row_df.loc[row_df.nodes.map(len) > 0]  # Filter out empty graphs

    row_df[["input", "code_embedding_mapping"]] = row_df.apply(
        lambda row: process_nodes_to_input_row(row, w2vmodel), axis=1)

    progress.update(main_task, advance=1)
    i += 1

    output_df = row_df if not df_init else pd.concat([output_df, row_df])
    df_init = True if not df_init else df_init

    if w2v_init:
        w2v_init = False

    if i % EXAMPLES_PER_SAVE == 0:
        output_df.to_pickle(output_path)
        w2vmodel.save('tmp/dataset/w2v/w2vmodel.wv')
        print(f"Saved dataset at {output_path}")

# Final save
w2vmodel.save('tmp/dataset/w2v/w2vmodel.wv')
output_df.to_pickle(output_path)
print(f"Final dataset saved at {output_path}")
...
```

Figure 3.9: Code snippet for transforming extracted CPGs into input graph embeddings. Functions are tokenized, Word2Vec embeddings are trained, and the resulting graphs are assembled and saved for training (Full code in Figure C7).

3.4 Base Model for Vulnerability Detection

To accurately detect vulnerabilities in real-world software code, a model must be capable of reasoning over both local and global semantic relationships, handling diverse structural patterns, and generalizing beyond superficial syntax. Graph-based learning has emerged as a promising approach to this challenge, particularly through Graph Neural Networks (GNNs) [Sca+09] that operate on structured representations of source code. Within this landscape, **Devign** [Zho+19] stands out as a foundational GNN model specifically designed for vulnerability detection in C/C++ source code.

VISION adopts Devign as its base model due to its effectiveness, interpretability, and alignment with the types of code graphs generated by the Joern toolchain [Whi25]. Devign has demonstrated consistent improvements over traditional deep learning models such as CNNs and RNNs, particularly when applied to datasets containing rich control flow and data dependency structures. By directly operating on Code Property Graphs (CPGs) [Yam+14], Devign is able to leverage a combination of program views to extract meaningful features for classification.

The primary strength of Devign lies in its ability to model graph-structured inputs and to learn representations that capture code semantics across multiple abstraction levels. The prediction pipeline is illustrated in Figure 3.10, showing how Devign processes graph-structured inputs to output binary vulnerability predictions.

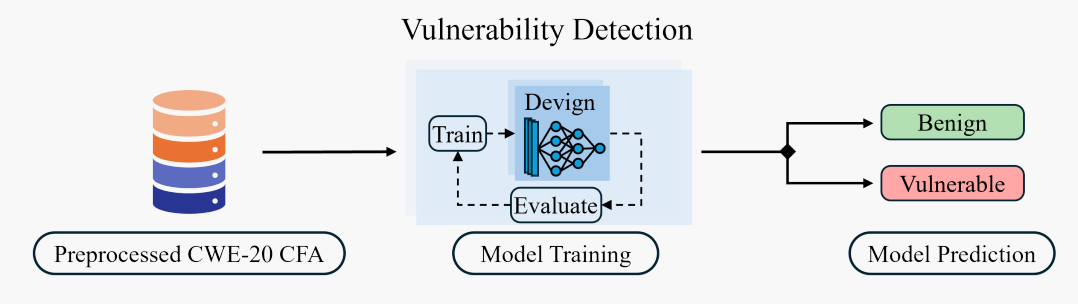


Figure 3.10: Vulnerability detection phase of the VISION framework. Preprocessed inputs from the CWE-20-CFA dataset are used to train and evaluate the Devign model. After training, the model classifies each graph as either vulnerable or benign,

3.4.1 Devign Model Architecture

The architecture of Devign consists of three main components, each contributing to a different stage of the graph learning process (see Figure 2.6):

1. **Graph Embedding Layer**: This module transforms nodes and edges of a code graph into continuous vector representations. Each node—representing a code token or symbol—is initialized with a learned embedding that captures its syntactic type and lexical identity. Edge types (e.g., `AST_CHILD`, `DFG_READS`, `CFG_NEXT`) are also encoded, allowing the model to preserve the semantics of different graph relations. The resulting graph is a heterogeneous structure enriched with token-level and structural embeddings, which are passed downstream for processing.

2. **Gated Graph Recurrent Unit (GGRU) Layers**: Devign employs gated message passing inspired by Gated Recurrent Units (GRUs) to iteratively propagate information across nodes. At each layer, node embeddings are updated by aggregating messages from neighbors using trainable gating functions. This allows the model to capture long-range dependencies in the graph, making it robust to variations in function length and complexity. The use of GGRUs also addresses issues such as oversmoothing and vanishing gradients, which commonly affect deeper graph architectures.

3. **Graph-Level Convolutional Module**: Once the node embeddings have been enriched through several rounds of message passing, Devign applies global pooling and convolutional operations to summarize the graph into a single vector. This representation captures the overall structural and semantic profile of the function. A final fully connected layer maps the graph-level embedding to a binary output indicating whether the function is vulnerable or benign.

The implementation of the Devign architecture is shown in Figure 3.11.

```python
class Devign(nn.Module):

    def __init__(self, gated_graph_conv_args, conv_args, emb_size: int):
        super(Devign, self).__init__()

        # Gate Graph Convolution Layer
        self.ggc = GatedGraphConv(**gated_graph_conv_args)

        # Convolutional layers
        self.conv1d_1 = nn.Conv1d(**conv_args["conv1d_1"])
        self.bn1 = nn.BatchNorm1d(conv_args["conv1d_1"]['out_channels'])
        self.conv1d_2 = nn.Conv1d(**conv_args["conv1d_2"])
        self.bn2 = nn.BatchNorm1d(conv_args["conv1d_2"]['out_channels'])

        # Activation layers
        self.relu1 = nn.LeakyReLU()

        # Fully connected layers
        fc1_size = gated_graph_conv_args["out_channels"] + emb_size
        fc1_size = self.get_conv_mp_out_size(fc1_size, conv_args["conv1d_2"], \
            [conv_args["maxpool1d_1"], conv_args["maxpool1d_2"]])
        fc2_size = gated_graph_conv_args["out_channels"]
        fc2_size = self.get_conv_mp_out_size(fc2_size, conv_args["conv1d_2"], \
            [conv_args["maxpool1d_2"], conv_args["maxpool1d_2"]])
        self.fc1 = nn.Linear(in_features=fc1_size, out_features=1)
        self.fc2 = nn.Linear(in_features=fc2_size, out_features=1)

        # Max pooling
        self.mp_1 = nn.MaxPool1d(**conv_args["maxpool1d_1"])
        self.mp_2 = nn.MaxPool1d(**conv_args["maxpool1d_2"])

        self.count_parameters()

    def forward(self, data) -> torch.Tensor:
        x, edge_index = data.x, data.edge_index

        # Gated Graph Convolution
        hidden = self.ggc(x, edge_index)

        # Concat GGC output with input embeddings
        concat = torch.cat([hidden, x], 1)
        concat_size = hidden.shape[1] + x.shape[1]
        concat = concat.view(-1, self.conv1d_1.in_channels, concat_size)

        # Path Z
        Z = self.mp_1(self.relu1(self.bn1(self.conv1d_1(concat))))
        Z = self.mp_2(self.bn2(self.conv1d_2(Z)))

        # Path Y
        hidden = hidden.view(-1, self.conv1d_1.in_channels, hidden.shape[1])
        Y = self.mp_1(self.relu1(self.bn1(self.conv1d_1(hidden))))
        Y = self.mp_2(self.bn2(self.conv1d_2(Y)))

        # Flatten
        Z = Z.view(-1, Z.shape[1] * Z.shape[-1])
        Y = Y.view(-1, Y.shape[1] * Y.shape[-1])

        # Final interaction
        res = self.fc1(Z) * self.fc2(Y)

        # Output
        sig = torch.sigmoid(torch.flatten(res))
        return sig
```

Figure 3.11: Definition of the `Devign` class and forward method. The model combines gated graph convolution with sequential convolutional layers, pooling, and fully connected prediction heads.

3.4.2 Integration with the VISION Framework

Within the VISION pipeline, Devign serves as the core classifier tasked with distinguishing between benign and vulnerable code samples. Each function—whether an original example from the PrimeVul CWE-20 subset or a generated counterfactual—is first transformed into a code property graph (CPG), embedded, and formatted into a graph data structure as described in subsection 3.3.4. These structured inputs are then fed into the Devign architecture, which processes them through its graph embedding and convolutional layers to produce a vulnerability prediction.

The integration is seamless: the graph inputs created from the preprocessing stage (containing node features and edge indices) are directly compatible with the PyTorch Geometric implementation of Devign. This modularity allows for flexible experimentation across different data splits, class ratios, and augmentation levels.

3.4.3 Training on Original and Counterfactual Pairs

A key component of VISION is the use of counterfactual training pairs. Unlike conventional training, where each example is treated independently, the training in VISION implicitly encodes **contrastive information**: by learning from both a function and its counterfactual counterpart, the model is encouraged to focus on discriminative features—those responsible for flipping the vulnerability label.
This setting is particularly effective in mitigating shortcut learning and overfitting to dataset-specific patterns. For example, two samples might share identical structure except for the inclusion of a single validation check; the model, when trained on both, is guided to assign importance to that precise difference. This forces the GNN to rely on semantically meaningful code constructs, such as control conditions, argument sanitization, or tainted data flows.

3.4.4 Loss Function and Optimization

Devign is trained using a binary classification loss function (typically binary cross-entropy), optimized using standard gradient descent techniques (e.g., Adam). The final classification layer outputs a scalar between 0 and 1, interpreted as the probability of vulnerability. Ground-truth labels are derived directly from the CWE-20-CFA dataset, with 0 representing benign and 1 representing vulnerable examples.

Formally, the loss for a batch of n samples is computed as:

$$L = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

where y_i is the true label and $\hat{y}_i$ is the model's predicted probability for the i-th input graph.

To ensure stability, early stopping and learning rate scheduling are employed. Each model variant (per dataset split) is trained independently, allowing for comparative analysis of how the training distribution affects model performance.

3.5 Explanation Module

While accurate vulnerability detection is essential, understanding why a model classifies code as vulnerable is equally critical—especially in high-stakes software security contexts. The Explanation Module in the VISION framework addresses this need by integrating post hoc interpretability tools that reveal which components of a source code graph contribute most to a model's decision. This module enhances both transparency and trust in the system's predictions and supports human-in-the-loop analysis by developers or auditors. Building upon the Devign model's node-level output structure, the explanation module leverages graph-based attribution techniques to highlight influential subgraphs, enabling semantic inspection of both benign and vulnerable predictions.

3.5.1 Motivation and Role of Explainability

While high predictive accuracy is a central goal in machine learning, it is not sufficient in safety-critical domains such as cybersecurity. In the context of software vulnerability detection, simply labeling a piece of code as *vulnerable* or *benign* is not enough—particularly when the consequences of that label may affect system reliability, code audits, compliance, or security patching. What is equally important is understanding the rationale behind the model's decision, especially when the input space is as complex and structurally rich as source code [Mou+24].

Graph Neural Networks (GNNs), though powerful, behave as inherently opaque models. Their layered message-passing operations and high-dimensional embeddings offer limited human interpretability. As a result, they often function as black-box classifiers [WMR18], even when trained on well-structured representations such as Code Property Graphs (CPGs). This lack of transparency poses significant challenges in real-world applications, where developers, auditors, or security analysts must validate model outputs, verify false positives, or understand the model's weaknesses.

Explainability, therefore, plays a dual role in the VISION framework. On one hand, it promotes trust by revealing which parts of the code graph most influenced the model's prediction. On the other, it supports model debugging and improvement by helping developers detect spurious correlations or unintended shortcuts learned during training. This is particularly critical in the presence of counterfactual examples: when two code functions differ minimally but receive opposite classifications, explanations can help determine whether the model is truly focusing on the semantically meaningful difference between them.

Additionally, explainability contributes to regulatory compliance and responsible AI practices. In high-assurance settings, being able to justify a model's output is a prerequisite for deployment. By embedding interpretability as a core module, VISION aligns with broader goals of AI transparency, verifiability, and human-centered oversight [Sha+22].

In the next subsections, the explanation technique used in VISION is detailed, the

types of attributions produced, and how these integrate with the underlying base model for graph-based vulnerability detection.

3.5.2 Explanation Technique: Illuminati Explainer

To address the challenge of explaining GNN-based predictions in the cybersecurity domain, the VISION framework incorporates **Illuminati** [HJH23], a domain-specific explainer designed for graph-structured data. Illuminati focuses on generating faithful, fine-grained attributions over the elements of a graph—specifically **nodes**, **edges**, and **node features**—providing interpretable subgraphs that highlight the regions most responsible for a prediction.

Illuminati builds upon the general concept of **mask-based explanation**. It introduces a learnable mask over the input graph that selectively highlights a subset of nodes and edges while preserving the model's prediction. The optimization objective is to maximize fidelity (i.e., the explanation should lead to the same prediction as the full graph) while minimizing the number of elements included, resulting in a sparse, semantically meaningful explanation.

Formally, given a graph $G = (V, E)$, a GNN model f, and a prediction $y = f(G)$, Illuminati learns a mask M over the nodes, edges, and features such that the masked graph G_M satisfies:

$$f(G_M) \approx f(G) \quad \text{and} \quad |G_M| \ll |G|$$

The mask M is trained using a differentiable loss function that penalizes both prediction divergence and mask size, thereby striking a balance between faithfulness and interpretability. The resulting explanation is a soft subgraph that can be thresholded or visualized to show which parts of the code graph are most influential to the model's decision.

The full explanation workflow of Illuminati is illustrated in Figure 3.12, highlighting how edge and feature attribution are used to generate interpretable node-level subgraphs.

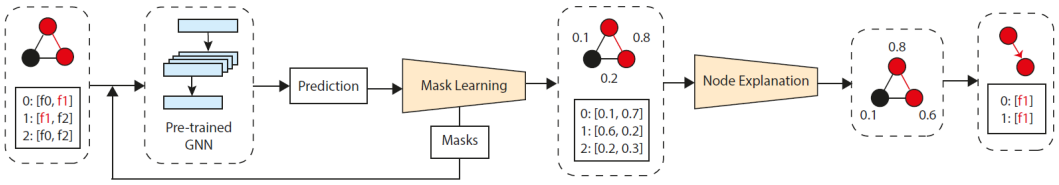


Figure 3.12: Workflow of the ILLUMINATI explanation framework. Given an input graph and a pre-trained GNN, the explainer first learns importance scores for edges and node attributes. These scores are then used to compute node-level attributions. Finally, the resulting subgraph explanation is derived by removing components with low attribution, yielding a concise and interpretable explanation of the model's decision. Adapted from [HJH23].

A key strength of Illuminati lies in its **domain awareness**. Unlike generic explainers, it is specifically designed to support graph structures common in software analysis, such as Code Property Graphs (CPGs). These CPGs include heterogeneous edge types (e.g., AST, CFG, DFG), making it critical that the explainer respects both syntax and data/control flow semantics.

In addition to fidelity and compactness, the method is model-agnostic and can be applied to a wide range of GNN architectures, including the Devign model used in our framework. Illuminati supports both graph-level and node-level predictions, making it well-suited to the task of code vulnerability detection, where the explanation must capture structural cues such as control flows or data dependencies.

Illuminati was evaluated in its original paper using tasks such as malware detection and intrusion detection, showing high-quality explanations with significantly improved Intersection-over-Union (IoU) scores compared to prior explainers like PGExplainer [Luo+20] and GNNExplainer [Yin+19]. Figure 3.13 shows a comparative explanation of a vulnerability case adapted from the original paper, highlighting the advantages of Illuminati in capturing both node and feature relevance.

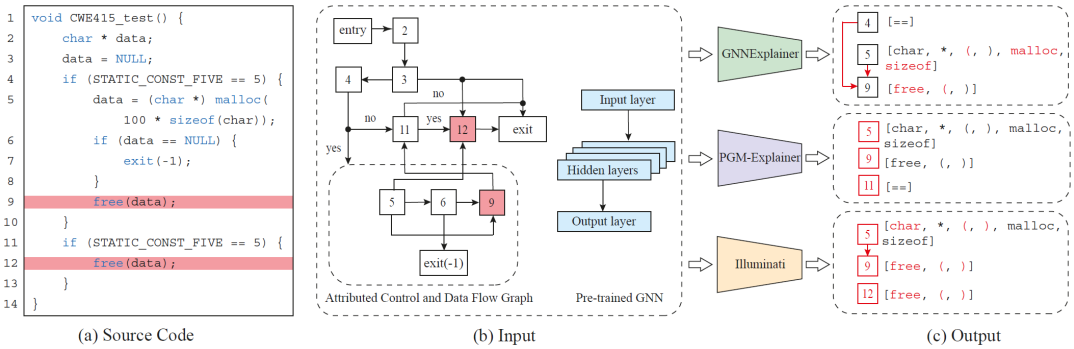


Figure 3.13: Explaining a sample code predicted as vulnerable using different explanation methods. (a) shows the source code containing a CWE-415 double free vulnerability. (b) depicts the attributed control and data flow graph used as model input, along with a pre-trained GNN. (c) presents the output of three explanation methods. GNNExplainer highlights edge relevance but cannot distinguish token context; PGM-Explainer identifies relevant nodes without attributing features; Illuminati identifies both key nodes and discriminative token-level features, offering a more precise explanation. Adapted from [HJH23].

In the context of VISION, Illuminati operates on the graphs generated during preprocessing, allowing explanations to be derived not only for original functions but also for their counterfactual counterparts. This enables an explanation-driven contrastive analysis, where differences in model behavior across paired inputs can be directly visualized and interpreted. Such capability is particularly valuable for inspecting whether the model correctly attends to the minimal vulnerability-flipping edits introduced during augmentation. This also helps expose reliance on potentially spurious code patterns and strengthens the trustworthiness of predictions.

Overall, Illuminati provides a principled and model-faithful way to open the black box of GNN predictions, enhancing the interpretability and auditability of the entire detection pipeline.

3.5.3 Attribution Modes and Interpretation

Illuminati produces rich, fine-grained attributions over the input graph components to explain the predictions of a Graph Neural Network (GNN). These attributions quantify how much each part of the graph—nodes, edges, and features—contributes to the final prediction. In the VISION framework, this attribution process enables the model to highlight precisely which parts of the code graph are responsible for classifying a function as vulnerable or benign.

Node, Edge, and Feature Attribution

Illuminati first learns the importance masks over nodes, edges, and node features, each offering a distinct perspective on model behavior:

- **Node Attribution** assigns a score to each node representing its influence on the prediction. Higher values indicate nodes that, if removed, would significantly affect the model output.
- **Edge Attribution** reveals which connections between nodes are structurally or semantically important (e.g., function calls, data dependencies).
- **Feature Attribution** identifies which specific code-level tokens or types (e.g., `malloc`, `free`, `NULL`) within a node's representation are decisive in the model's reasoning.

By combining these modes, the explainer not only isolates important statements but also clarifies why they matter in the specific context of the prediction.

Subgraph-Based Interpretations

To support human understanding and semantic clarity, the raw attribution scores are used to construct interpretable subgraphs. These subgraphs summarize the influential parts of the input and are classified into three main categories:

- **Positive Subgraph**: A positive subgraph is constructed by incrementally adding the most important nodes until the model's prediction aligns with the ground truth. This reveals which key components are sufficient to support a correct prediction.
- **Negative Subgraph**: A minimal set of nodes that, formed by progressively removing the least important nodes until the model's prediction flips. This highlights which components are essential for the prediction and reveals the model's sensitivity to the absence of specific, seemingly unimportant structures.
- **Optimal Subgraph**: A compact explanation that balances fidelity and interpretability by selecting nodes and edges that maximize prediction retention while minimizing complexity.

These explanation types allow analysts to inspect the prediction from both additive and subtractive perspectives—what contributes most to the prediction, and what can be removed before it changes.

Interpretability Through Attribution Scores

Each graph element is assigned a continuous score between 0 and 1, indicating its relevance to the prediction. These scores are typically visualized using color gradients or ranked lists (explored in detail in section 3.6), but they can also be interpreted by thresholding or aggregating scores across graph layers.

For example, in a CWE-20 vulnerability involving a missing input check, a node representing the unchecked parameter may receive a high attribution score, while surrounding control statements receive much lower values. This provides targeted insight into the exact vulnerability trigger.

These attribution modes form the conceptual bridge between the Devign classifier and the human-readable explanations provided by the VISION framework. The next section describes how this integration is realized technically and how the explainability module interacts with Devign's outputs.

3.5.4 Illustrative Example: Subgraph-Based Interpretations

To better understand how the Illuminati explanation module works in practice, this section presents a complete, illustrative walkthrough of the subgraph-based explanation modes applied to a simple function vulnerable to CWE-20: Improper Input Validation. The goal is to demonstrate how different types of subgraphs—positive, negative, and optimal—can be derived from attribution scores and used to support the interpretation of the model's predictions.

Explanation Setup

Figure 3.14 shows the function being analyzed. It copies user input into a local buffer using `strcpy`, but only after checking that the input is not `NULL`. Despite the presence of this validation, the use of `strcpy` without additional length checks constitutes an input validation weakness. This makes the function a canonical example of CWE-20.

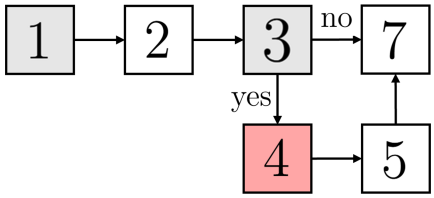
Illuminati assigns an importance score to each node in the function's code graph, reflecting its relative contribution to the vulnerability prediction made by the Devign model. The most important nodes—specifically the sink operation (`strcpy`) and the input validation condition—receive the highest scores. These scores are summarized in the accompanying attribution table and visualized in a graph structure.

```c
void process(char *input) {
    char buffer[100];
    if (input != NULL) {
        strcpy(buffer, input);
        printf("Processed input.\n");
    }
    printf("Function done.\n");
}
```

(a) Source Code

Node ID	Node Code Fragment	Node Importance Score
1	`void process(char *input)`	0.85
2	`char buffer[100];`	0.32
3	`if (input != NULL)`	0.74
4	`strcpy(buffer, input);`	0.92
5	`printf("Processed input.");`	0.08
7	`printf("Function done.");`	0.10

(b) Node Importance Scores

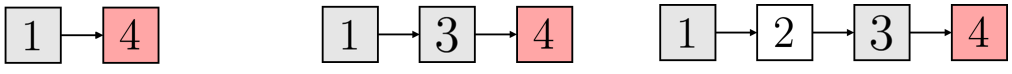


(c) Node Graph Representation

Figure 3.14: Explanation of a vulnerable CWE-20 function using the Illuminati framework. (a) Source code for the function; (b) statement-level node importance scores; and (c) graph representation. The high-importance node triggering the vulnerability is highlighted in red.

Interpreting Subgraphs

Figure 3.15 visualizes each of these subgraph types. The nodes are shaded to reflect their relative attribution scores, and their inclusion is selected based on how they affect the model's output confidence.



(a) Positive Subgraph (b) Negative Subgraph (c) Optimal Subgraph

Figure 3.15: Example of different explanation subgraphs for a code function: (a) Positive subgraph, (b) Negative subgraph and (c) Optimal subgraph. Highlighted nodes reflect increasing attribution scores.

Model Confidence Evaluation

To quantify the fidelity of each subgraph, Table 3.3 reports the confidence scores produced by the Devign model when evaluating each version of the graph. As expected, the full graph produces the highest confidence, while subgraphs yield slightly lower confidence but still retain the correct label in most cases. The optimal subgraph achieves nearly equivalent confidence while using significantly fewer nodes.

Table 3.3: Confidence scores obtained when evaluating different subgraphs generated by the Illuminati explainer on a vulnerable CWE-20 code example. The full graph includes all statements; the positive subgraph is the minimal set required to preserve the prediction; the negative subgraph is the smallest subset whose removal causes the label to flip; the optimal subgraph balances fidelity and compactness.

Subgraph Type	Included Nodes	Confidence
Full Graph	1-7	0.98
Positive Subgraph	1, 4	0.87
Negative Subgraph	1, 3, 4	0.92
Optimal Subgraph	1, 2, 3, 4	0.95

3.5.5 Integration with Base Model

The integration of the Illuminati explainer into the VISION framework is enabled by the architectural properties of the base model (Devign). Since Devign operates on Code Property Graphs (CPGs) and produces node-level embeddings through gated message passing, it naturally supports fine-grained attribution analysis over both node and edge components—exactly the interface required by Illuminati. As illustrated in Figure 3.16, the explainability module operates after prediction to provide semantic insight into the model's decision, enabling both node-level attribution and subgraph extraction.

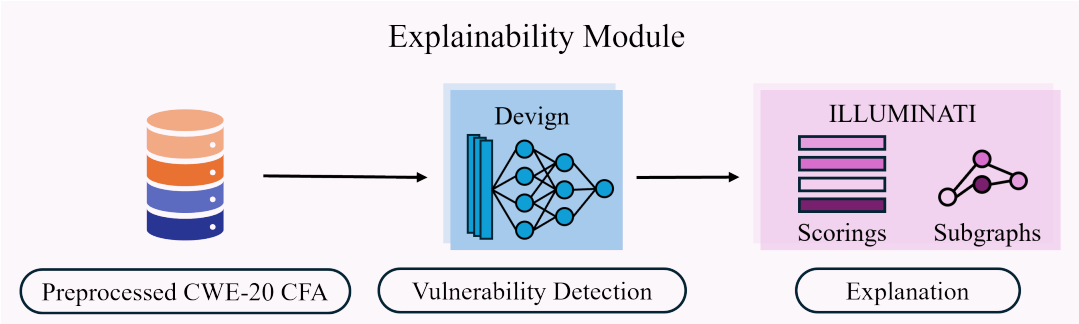


Figure 3.16: Overview of the Explainability Module within the VISION framework. Once the Devign model generates predictions over the preprocessed CWE-20-CFA dataset, the Illuminati explainer is used to derive meaningful explanations. It produces relevance scorings over graph components and extracts minimal subgraphs that highlight the key elements responsible for the classification, enabling transparent and interpretable vulnerability analysis.

Illuminati functions as a post-hoc explainer: it does not alter the architecture or training of Devign. Instead, once the Devign model is trained and capable of producing vulnerability predictions for graph-structured inputs, Illuminati is applied to analyze individual predictions. Specifically, it takes as input:

- The original input graph $G = (V, E)$, including node features and edges.
- The prediction $f(G)$ made by Devign.
- The internal structure of the model (e.g., GNN layers and message passing dynamics) to enable gradient-based mask optimization.

This modular design ensures seamless compatibility between the explanation module and any Devign instance trained on the CWE-20-CFA dataset. No architectural changes or retraining are necessary. The explainer operates directly on the `torch_geometric.data.Data` objects passed to Devign, making use of node embeddings, graph connectivity, and gradient information.

Another key strength of this integration lies in its ability to support contrastive explanations. Since the VISION framework includes both original and counterfactual versions of each code function, explanations can be generated for both. This allows for direct comparison of attribution maps across paired samples with opposite labels—an important capability for verifying whether the model is attending to the intended vulnerability-inducing change introduced during augmentation.

Moreover, because the node ordering and feature vectors are preserved from preprocessing, the attribution results are easily interpretable in terms of the original source code tokens. This tight linkage between data representation, model prediction, and explanation makes the Devign–Illuminati pairing especially effective for use in the software vulnerability detection domain.

3.6 Visualization module

To bridge the gap between model predictions and human interpretability, this project includes a dedicated visualization module that presents the internal decision-making process of the vulnerability detector in a clear, interactive, and interpretable way. The module enables detailed inspection of individual code examples by overlaying attribution-based explanations—produced by the Illuminati explainer—onto both the source code and its corresponding graph structure. Its main purpose is to qualitatively validate whether the learned attributions reflect semantically meaningful structures and whether the improvements introduced by counterfactual data augmentation translate into more interpretable and trustworthy model behaviors. While this interface is not the core algorithmic innovation of the framework, it plays a crucial role in human-in-the-loop evaluation, debugging, and explanation validation.

3.6.1 Overview and Purpose

The visualization module developed in this project serves as a critical interpretability layer for understanding the behavior of the vulnerability detection model. While the core contributions of the VISION framework lie in its robust training pipeline and graph-based explanation techniques, this module acts as a practical interface for exploring how model predictions are made and how explanation scores evolve across examples.

Its primary purpose is to support **human-in-the-loop analysis**, allowing researchers and practitioners to verify whether the model attends to semantically meaningful patterns in the code. In particular, the module helps assess whether the improvements in generalization and robustness—introduced through counterfactual data augmentation—are reflected in more accurate and focused attribution maps.

The visual interface integrates multiple components of the framework: the trained Devign model, the Illuminati explainer, and a rendering engine that overlays importance scores on both the original source code and the graph representation. This integration enables the user to simultaneously inspect:

- The model's **predicted label** and **confidence**,
- The **token-level attribution map**, and
- The **graph-level explanation structure**.

Through this interface, the visualization module not only aids in qualitative validation of the model's decisions, but also supports broader experimentation, such as comparing explanation patterns across different training configurations (e.g., with and without counterfactuals), or between original and counterfactual versions of the same function.

Although not a standalone contribution, the module has proven invaluable for confirming key hypotheses throughout the research: that better robustness leads to better explainability, and that these improvements can be intuitively observed through well-aligned, human-interpretable attributions.

3.6.2 Interface Design and Functionality

The visualization module is designed to offer a clean, dual-perspective view of model predictions and their explanations, combining **natural source code structure** with **graph-based reasoning**. Its interface is modular and composed of synchronized panels that allow users to interpret attribution scores from multiple angles. This helps ensure that explanations are not only accurate from the model's perspective but also intuitive for human analysis.

The visualization module was implemented using the **Gradio** Python library [Gra25], which offers a lightweight yet powerful interface to interact with machine learning models. The module runs locally on `localhost:127.0.0.1:7068`, but it also allows connections from external devices within the same network,

enabling collaborative and remote usage.

At first glance, the interface provides a clean layout with clearly separated functional zones (Figure 3.17). On the left-hand side, the user finds a large text box labeled **Input Source Code**, which serves as the main display for the function being analyzed. The user can interact with this section in two ways:

- Pressing the **Random** button loads a randomly selected source code example from the dataset.
- Alternatively, the user can load a specific example by entering its unique identifier (referred to as *Example ID*) into the corresponding text field and then pressing the **Select** button.

Each source code example in the system is indexed using a **consistent ID** shared across the original and counterfactual versions, which allows for aligned comparisons and navigation.

Once an example is loaded, the **Submit** button initiates a full pipeline execution: the example is passed through the trained model, explanations are computed, and visual outputs are rendered. To reset the session or clear previous outputs, the **Clear** button can be used to instantly wipe the interface.

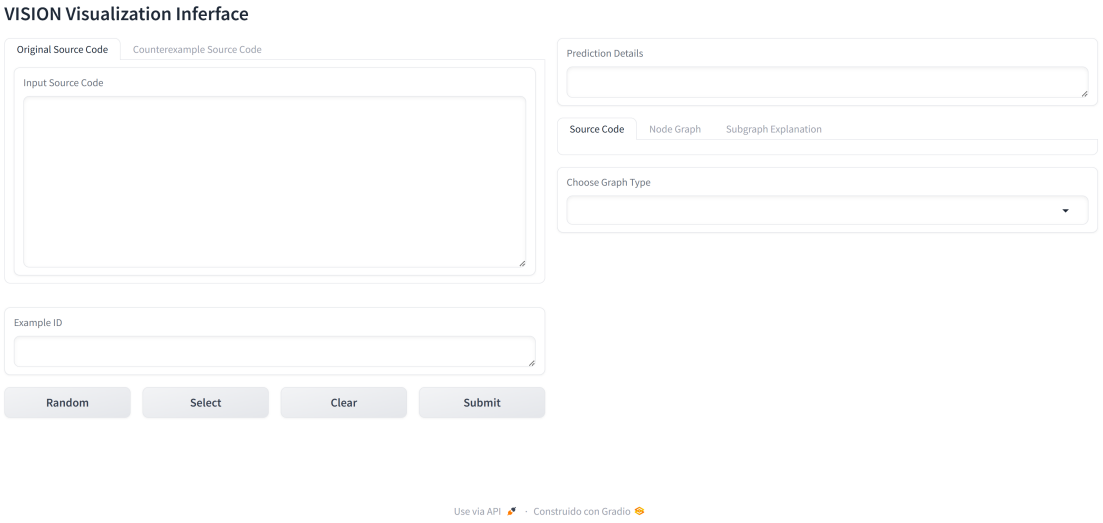


Figure 3.17: Initial view of the VISION visualization interface, implemented in Gradio. On the left, users can display code by either selecting a random example or providing an `Example ID`. Once loaded, pressing `Submit` runs the example through the model and generates explanations. The `Clear` button resets the view.

After submission, the right-hand panel of the interface becomes active and displays:

- The model's predicted label under **Prediction Details**.
- A tab-based layout offering three visualization modes: **Source Code**, **Node Graph**, and **Subgraph Explanation**.
- A dropdown menu to toggle between different types of graph visualizations (e.g., AST, CPG).

Source Code View: Attribution-Aligned Text Highlighting

Once a source code example is submitted, the **Source Code** tab displays a visually annotated version of the input function. Each token in the function is colored using a heatmap gradient that reflects its attribution score, as computed by the Illuminati explainer. This allows users to quickly assess which code fragments the model deemed most influential in forming its prediction.

As shown in Figure 3.18, the source code on the left corresponds to the raw function input. On the right side, the same code is presented with attribution highlights. Tokens with higher importance appear in more intense red or purple shades, while neutral or low-importance regions remain uncolored or lightly shaded.

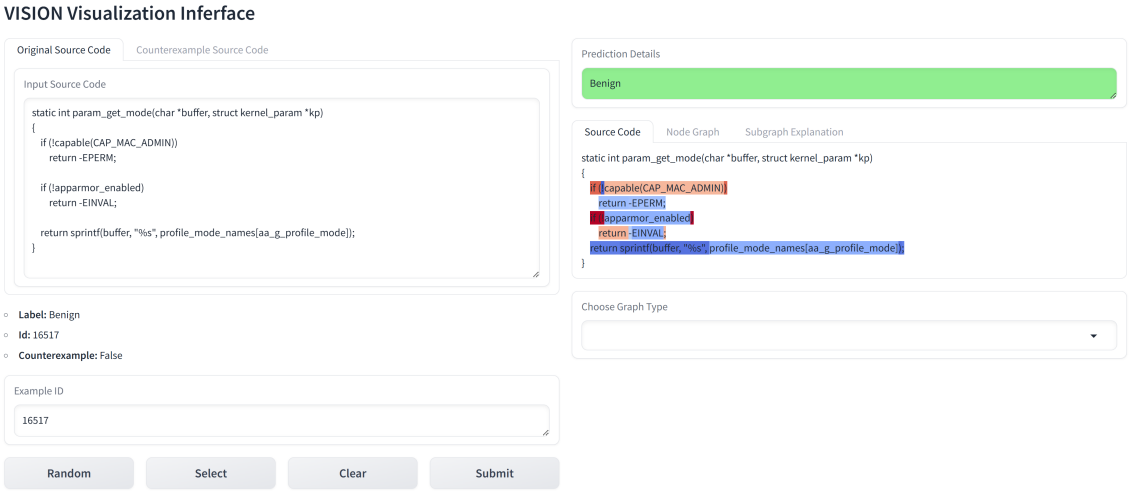


Figure 3.18: Source Code view for an original benign example. The interface highlights the most influential tokens based on node-level attribution scores. In this case, the model correctly predicts a benign label, with explanation heatmaps focusing on control flow and input-related conditions.

In this specific example, the function is labeled as benign, and the model correctly classifies it as such. The highlights emphasize conditional statements like `if (!capable(...))` and `if (!apparmor_enabled)`, as well as the final return line involving `sprintf(...)`. These elements reflect key security-related checks and data flow operations—suggesting that the model has learned to identify critical validation structures.

This view is especially useful for identifying whether the model's reasoning aligns with human intuition. In robust models, benign examples typically show attribution spread across validation logic, while vulnerable examples often highlight unsafe sink operations (e.g., `strcpy`, unchecked input usage). This alignment becomes a key element for trusting the model in practical applications like secure code review.

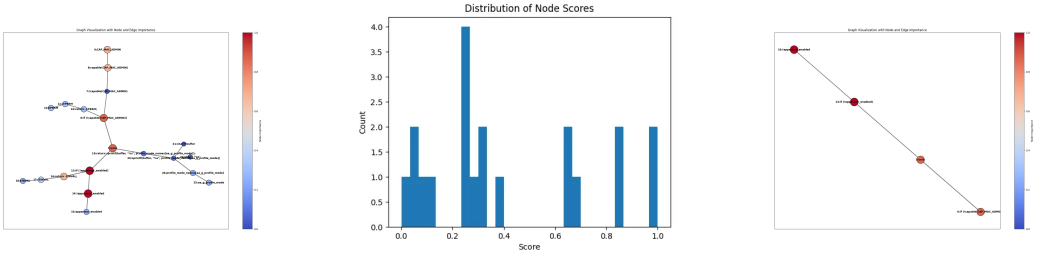
Graph View and Subgraph Interpretation

In addition to the token-highlighted source code, the visualization module offers a graph-level representation of the function under analysis. This view is available in the **Node Graph** and **Subgraph Explanation** tabs and visualizes the Code

Property Graph (CPG) as a node-edge structure enhanced with importance scores. Each node corresponds to a program element (e.g., statements, variables), and edges represent syntactic or semantic relations such as control or data dependencies.

Figure 3.19 illustrates the three main components shown in this part of the interface:

- **(a) Full Node Graph**: Every node is color-coded based on the attribution score output by the Illuminati explainer. The warmer the color (toward red), the more influential the node is in the model's decision. This allows users to visually assess how explanations are spatially distributed across the graph.
- **(b) Score Distribution Histogram**: A histogram showing the distribution of node attribution scores provides a compact summary of how influence is allocated. This can help spot whether the model relies heavily on a few dominant nodes or spreads its decision basis across several factors.
- **(c) Positive Subgraph**: A reduced version of the graph consisting only of the top-attribution nodes necessary to recover the correct prediction. This subgraph offers a compact and interpretable summary, isolating the critical decision-making region of the model.



(a) Full Node Graph (b) Node Score Distribution (c) Positive Subgraph

Figure 3.19: Different views of the graph-level explanation interface. (a) Full node graph visualization with attribution-based coloring. (b) Distribution of node importance scores from the Illuminati explainer. (c) Positive subgraph showing the minimal set of highly influential nodes sufficient to recover the correct prediction.

This tri-panel design reinforces the interpretability by:

1. Linking attribution heatmaps directly to the graph structure.
2. Quantifying influence distribution via summary plots.
3. Showing how smaller explanatory subgraphs can preserve fidelity to the full model prediction.

Additionally, the interface allows users to toggle between various graph perspectives using the *Choose Graph Type* dropdown. Available views include: AST (Abstract Syntax Tree), CFG (Control Flow Graph), Full Code Property Graph, Positive Subgraph, Negative Subgraph and Optimal Subgraph.

Original vs Counterfactual Exploration

A key feature of the VISION interface is its seamless support for comparing the behavior of the model on original and **counterfactual examples**. This functionality enables users to interactively explore how subtle semantic changes impact both the model's predictions and the resulting attribution patterns.

Users can toggle between the two versions of a code sample via the tabbed header at the top of the input display panel, labeled *Original Source Code* and *Counterexample Source Code*. When this second tab is selected, all subsequent interface actions—such as selecting a random example, entering an ID, or submitting for explanation—are applied directly to the counterfactual variant. The Figure 3.20 shows the source code view for the counterfactual example.

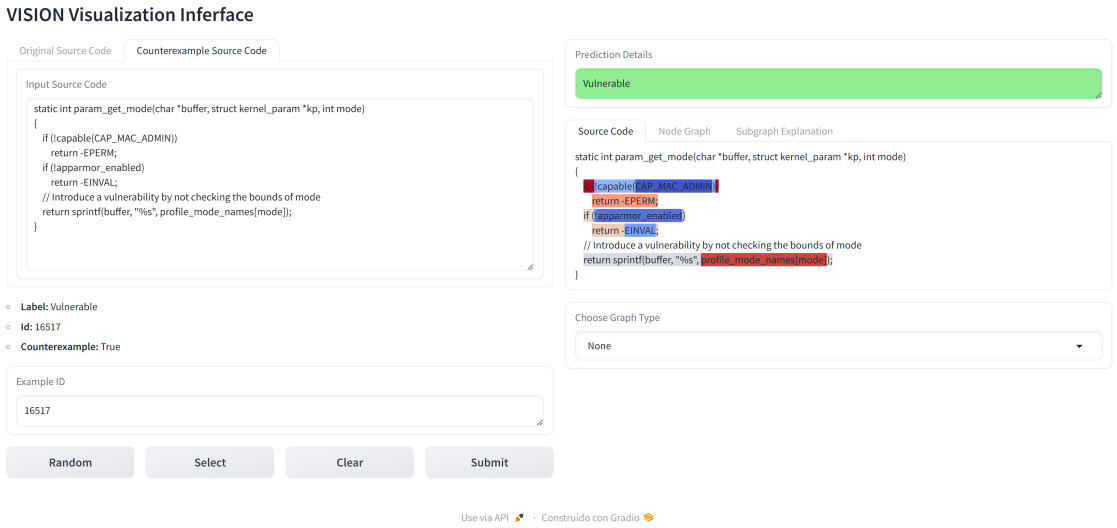
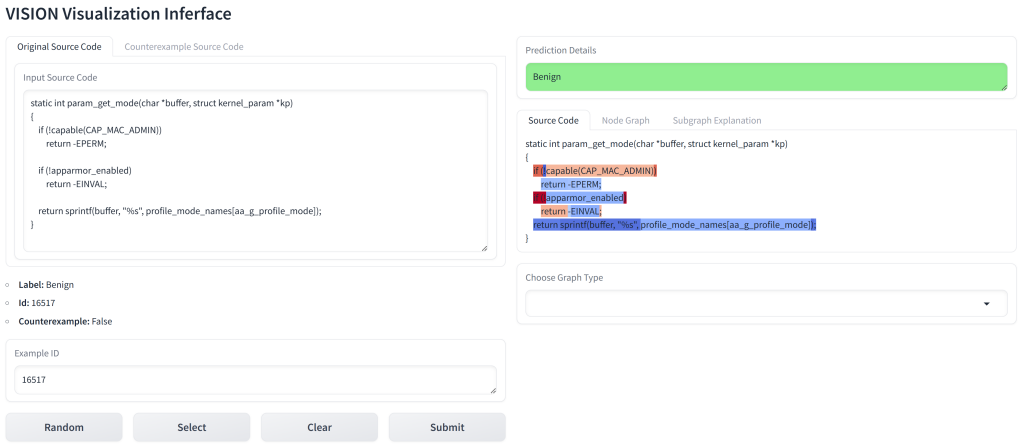


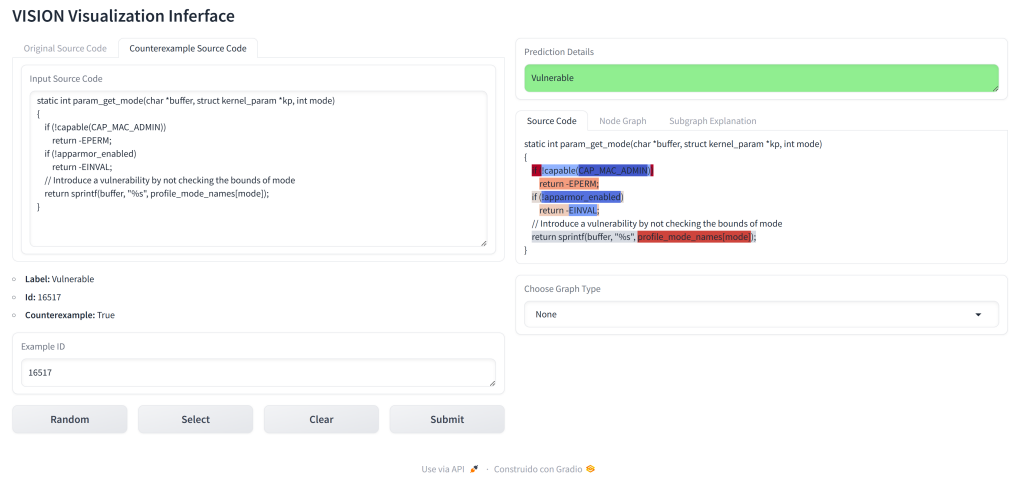
Figure 3.20: Counterexample source code view of the VISION visualization interface, implemented using Gradio. The interface allows seamless switching between original and counterfactual examples via tabs. In this view, a CWE-20 counterfactual is selected and submitted for analysis. The right panel displays the model prediction and highlights token-level attributions using a color scale that reflects node importance. This enables direct comparison between functionally related variants, improving transparency and interpretability.

Similarly, switching back to the original tab automatically restores the original version of the function, allowing quick back-and-forth inspection. The example ID remains consistent across both views, since counterfactuals are paired with their originals using shared identifiers. This design allows users to maintain context when analyzing the semantic and structural differences between both variants of the same function.

Figure 3.21 illustrates this functionality by displaying both the original benign example (top) and its minimally modified vulnerable counterfactual (bottom). The visualization panel highlights attribution differences and prediction shifts, providing concrete visual evidence of how counterfactual-based augmentation strengthens the model's semantic understanding.



(a) Original Source Code View – The benign example is correctly identified, with key validated checks highlighted.



(b) Counterexample Source Code View – The augmented vulnerable version omits validation and triggers the vulnerability detection.

Figure 3.21: Comparison of original and counterfactual examples within the VISION Visualization Interface. Users can easily switch between the two views using the tab selector above the source code input. The interface supports full interaction for either view—including random selection, ID-based lookup, and explanation generation—making it a practical and intuitive tool for exploring how small code changes affect model behavior and attribution.

3.7 Summary of Innovations

This final section concludes the technical description of the VISION framework by revisiting its **core innovations**. Throughout the chapter, each module has been detailed independently—from dataset generation to explainability and visualization—but here the emphasis shifts to the overall contribution of the framework as a cohesive and impactful research solution.

The VISION framework brings together a set of carefully designed components to address critical limitations in GNN-based source code vulnerability detection, particularly those related to data quality, spurious correlations, and

explainability. Each module—dataset construction, model training, explanation, and visualization—has been developed to work both independently and cohesively within a unified architecture. This section summarizes the major innovations introduced throughout this chapter and reflects on their collective impact.

Motivation and Challenges addressed

Traditional approaches to vulnerability detection using Graph Neural Networks have demonstrated potential but remain hindered by unreliable datasets, superficial feature learning, and limited interpretability. Models trained on noisy or imbalanced data often overfit to irrelevant patterns, reducing generalization. Moreover, the lack of transparent reasoning limits adoption in security-critical applications, where understanding the basis of a decision is as important as the prediction itself.

Key Contributions

To overcome these challenges, the VISION framework introduces several key innovations:

- **Counterfactual Data Augmentation via LLMs.** A novel strategy that generates semantically valid, minimally modified code variants using Large Language Models. These counterfactuals flip the vulnerability label of original examples and expose the model to meaningful variation, mitigating shortcut learning.
- **CWE-20-CFA Benchmark.** An augmented benchmark dataset specifically targeting the CWE-20 vulnerability class. Comprising over 27,000 samples, including carefully validated counterfactuals, CWE-20-CFA enables robust training and evaluation under balanced and realistic conditions.
- **Robust and Interpretable Detection.** A Devign-based GNN classifier is trained using the augmented data to enhance resilience against spurious features. The integration of the Illuminati explainer further enables fine-grained, subgraph-level attributions that clarify the internal reasoning of the model.
- **Interactive Visualization Module.** A lightweight but powerful interface was developed to support human-in-the-loop model inspection. It allows users to visualize node importance, graph structures, and contrast original vs counterfactual predictions, offering insights into how the model's behavior evolves with data augmentation.

Impact Overview

Together, these contributions deliver a scalable, extensible, and interpretable approach to secure software analysis using machine learning. VISION not only improves predictive performance and robustness but also contributes practical tools and benchmarks that support reproducibility and further research. By combining augmentation, explanation, and visualization, this framework demonstrates how thoughtful design can significantly enhance the trustworthiness of AI models in critical cybersecurity domains.

Chapter 4

Experimental Evaluation

To validate the effectiveness of the VISION framework, an extensive **experimental evaluation** is performed, analyzing model performance across multiple data augmentation configurations. The evaluation encompasses both conventional classification metrics and diagnostic indicators centered on **robustness** and interpretability. Specifically, the experimental design aims to assess whether counterfactual data augmentation mitigates spurious correlations, improves generalization, and enhances the semantic alignment of explanations in GNN-based vulnerability detection.

Performance is measured across a series of training benchmarks derived from the CWE-20-CFA dataset, using a consistent and balanced test set to ensure fair comparison. In addition to **standard predictive metrics**—accuracy, precision, recall, and F1-score—the evaluation includes robustness indicators such as pairwise accuracy, worst-group accuracy, and latent embedding analysis, as well as **explanation-focused metrics** including intra-class attribution variance, inter-class attribution distance, and a novel node score dependency measure.

4.1 Experimental Setup

4.1.1 Benchmark Construction and Splitting Strategy

To assess the impact of counterfactual data augmentation, a suite of benchmark datasets was derived from the **CWE-20-CFA** corpus introduced in 3.3.3. Each benchmark is configured with a fixed number of training samples and a constant 1:1 class balance (benign vs. vulnerable). However, the composition of original versus counterfactual examples varies across configurations, ranging from 100% original to 100% counterfactual, in 10% increments. This allows a controlled analysis of the contribution of each data type.

The splitting process begins with the partitioning of the CWE-20-CFA dataset using unique function identifiers to avoid overlap between related samples. A stratified 80/10/10 train-validation-test split is applied over the ID pool, ensuring that each original–counterfactual pair resides exclusively in one of the splits. To maintain fairness and comparability, the test set is held constant across all configurations and consists of perfectly paired examples (i.e., each original is matched with its

counterfactual), balanced across the two classes. The code fragment in Figure 4.1 outlines the process for building these datasets from a balanced CWE-20-CFA pool.

```
# Load processed datasets
print("Data Loading")
dataset_df = pd.read_pickle('datasets/PrimeVul/CWE-20_CFA.pkl')

# Step 1: Create balanced train/val/test split
train_df, val_df, test_df = group_train_val_test_split(
    dataset_df, test_size=0.1, val_size=0.1, random_state=SEED)

# Step 2: Fix the test set (used for all splits)
fixed_test_df = test_df.copy()
test_dataset = DevignDataset(fixed_test_df)
test_loader = test_dataset.get_loader(PROCESS["batch_size"], shuffle=False)

# Step 3: Build benchmarks
benchmark_datasets = {}
benchmark_splits = [(100-i, i) for i in range(0, 110, 10)]

for orig_pct, adv_pct in benchmark_splits:
    key = f'{orig_pct}_{adv_pct}'
    benchmark_train_df = create_balanced_symmetric_benchmark_split(
        train_df, orig_frac=orig_pct/100, size=len(train_df), random_state=SEED)
    benchmark_valid_df = create_balanced_symmetric_benchmark_split(
        val_df, orig_frac=orig_pct/100, size=len(val_df), random_state=SEED)

    benchmark_datasets[key] = {
        'train': benchmark_train_df,
        'valid': benchmark_valid_df
    }
```

Figure 4.1: Code for generating benchmark datasets with controlled ratios of original and counterfactual examples using the CWE-20-CFA corpus. This enables fine-grained evaluation of robustness and generalization by training under different augmentation regimes. The test set remains unchanged across all benchmarks.

Within each training split, examples are randomly sampled to match the desired original-to-counterfactual ratio. When needed, per-class upsampling is applied to ensure balance and equal sample counts across splits. Importantly, this methodology ensures that observed performance changes across configurations reflect differences in data composition, not volume or label imbalance. Figure 4.2 presents the benchmark configurations, highlighting the proportion of original, counterfactual, and upsampled examples in each case.

4.1.2 Preprocessing Pipeline and Feature Construction

The input to the base model consists of attributed graphs constructed from source code functions via Code Property Graphs (CPGs)n [Yam+14]. Each function, whether original or counterfactual, is parsed into its corresponding CPG using Joern [Whi25]. These graphs integrate multiple code representations, including ASTs, CFGs, and DFGs, enabling a comprehensive structural and semantic analysis of the function's behavior.

Once parsed, nodes in the CPG are ordered, filtered, and tokenized. Each node is then represented as a concatenation of its type and averaged token embedding. Edges are constructed based on AST relationships, resulting in input graphs suitable for the GNN architecture. This preprocessing is applied consistently across all datasets.

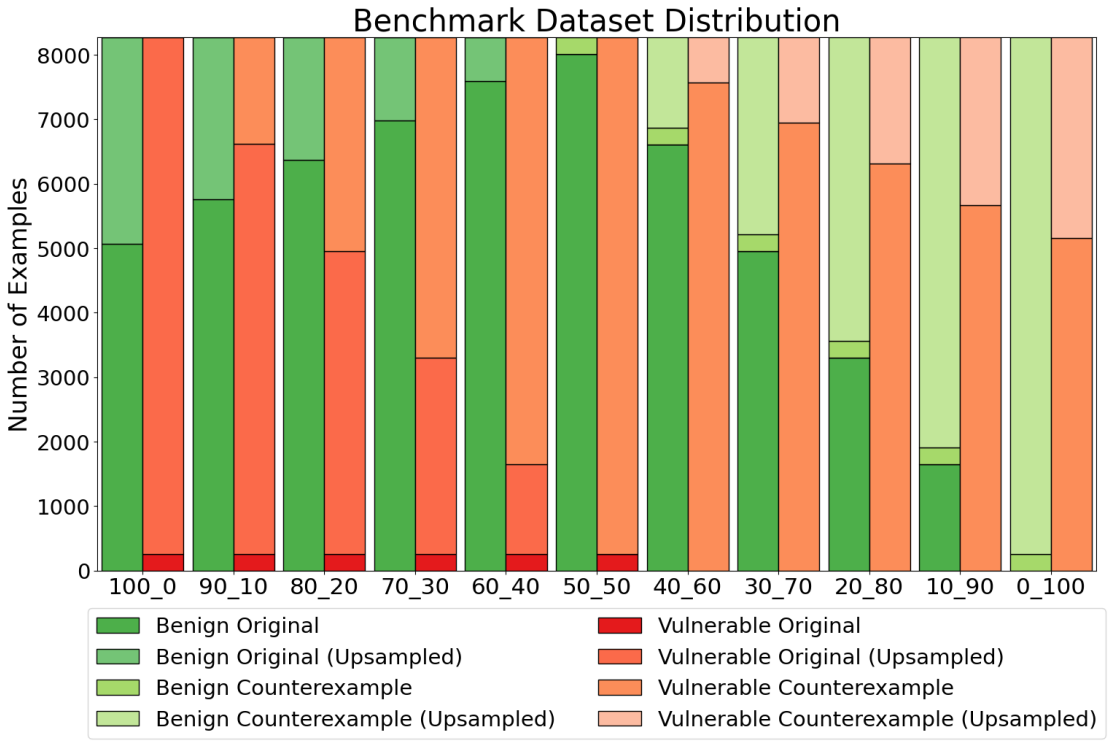


Figure 4.2: Benchmark dataset distribution by class (benign vs. vulnerable) and data source (original vs. counterfactual). Each bar stack shows the number of examples for benign and vulnerable samples (including the upsampled ones), and the split between original, counterfactual

4.1.3 Model Architecture and Training Configuration

All experiments are conducted using the Devign architecture [Zho+19], introduced in section 3.4. Each benchmark configuration is trained separately under an identical hyperparameter setup to ensure valid comparison and eliminate confounding effects due to training variability.

Table 4.1: Training hyperparameters used for all experiments with Devign. These settings are held constant across benchmarks to ensure a fair comparison.

Hyperparameter	**Value**
Optimizer	Adam
Learning Rate	5e-4
Batch Size	32
Epochs	100
Early Stopping (F1)	10
Weight Decay	1e-5
Loss λ	1e-6

The full list of training hyperparameters is presented in Table 4.1. These include the optimizer, learning rate, batch size, and early stopping criterion based on validation F1-score. A relatively small weight decay and loss lambda were employed to prevent over-regularization and allow the model to converge stably across diverse data splits.

Model architecture parameters used for all experiments. Includes the gated graph convolution configuration and convolutional layers of Devign.

```
DEVIGN = {
    "learning_rate" : 5e-4,
    "weight_decay" : 1e-0,
    "loss_lambda" : 1e-6,
    "model": {
        "gated_graph_conv_args": {
            "out_channels" : 200,
            "num_layers" : 6,
            "aggr" : "add",
            "bias": True},
        "conv_args": {
            "conv1d_1" : {
                "in_channels": 205,
                "out_channels": 50,
                "kernel_size": 3,
                "padding" : 1
                },
            "conv1d_2" : {
                "in_channels": 50,
                "out_channels": 20,
                "kernel_size": 1,
                "padding" : 1
                },
            "maxpool1d_1" : {
                "kernel_size" : 3,
                "stride" : 2
                },
            "maxpool1d_2" : {
                "kernel_size" : 2,
                "stride" : 2
                }
        },
        "emb_size" : 101
    }
}
```

Training hyperparameters including number of epochs, early stopping criteria, and batch configuration. Used consistently across all benchmark evaluations.

```
PROCESS = {
    "epochs" : 100,
    "patience" : 10,
    "batch_size" : 32,
    "dataset_ratio" : 0.2,
    "shuffle" : False
}
```

Figure 4.3: Configuration setup used for all benchmark evaluations. Top: Devign model architecture including Gated Graph Convolution and 1D convolutional layer parameters. Bottom: Training loop parameters including number of epochs, batch size, and early stopping. These were kept consistent across all original / counterfactual dataset splits to ensure fair comparison.

In addition to high-level training parameters, the full Devign configuration—including the architecture of the Gated Graph Convolution (GGC) and 1D convolutional layers—is shown in Figure 4.3. This Python dictionary represents the actual configuration used in the pipeline implementation, ensuring reproducibility and offering visibility into the architectural design.

The model training process is repeated independently for each benchmark configuration to ensure isolated evaluation. As shown in Figure 4.4, for every original/counterfactual ratio, a new instance of the Devign model is initialized and trained using a binary classification objective. The loop incorporates loss

```
...
for key, benchmark_df in benchmark_datasets.items():
        # For each benchmark obtain train and validation splits:
        train_dataset = DevignDataset(benchmark_df['train'])
        val_dataset = DevignDataset(benchmark_df['valid'])

        # Data Loaders
        train_loader = train_dataset.get_loader(PROCESS["batch_size"], \
            shuffle=False, use_sampler=False)
        val_loader = val_dataset.get_loader(PROCESS["batch_size"], \
            shuffle=False, use_sampler=False)

        # Model
        model = Devign(...)

        # Loss Function
        loss_fc = lambda o, t: F.binary_cross_entropy_with_logits(o, t) + \
            F.l1_loss(o, t) * loss_lambda

        # Optimizer
        optimizer = optim.Adam(model.parameters(), lr=learning_rate, \
            weight_decay=weight_decay)

        # Use learning rate scheduler
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, \
            mode='max', patience=5, factor=0.5)
        ...
        for epoch in range(PROCESS['epochs']):
            ...
            # Set model to training mode
            model.train()

            for i, batch in enumerate(tqdm(train_loader)):
                input = batch["input"].to(DEVICE)  # Move batch to GPU
                logit = model(input)  # Forward pass

                # Compute loss
                loss = loss_fc(logit, input.y)

                # Reset gradients
                optimizer.zero_grad()

                # Backpropagation
                loss.backward()

                # Gradient clipping
                nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

                # Perform parameter update
                optimizer.step()

                # Apply threshold to determine binary predictions
                preds = (logit >= THRESHOLD).float().cpu().numpy()
                acc = binary_accuracy(torch.tensor(preds), input.y.cpu())
                ...

            ...
            # Evaluate on validation set
            val_metrics = eval_model(model, val_loader, threshold=THRESHOLD)
            ...
```

Figure 4.4: Training loop used to train the Devign model across different augmentation benchmarks. Includes optimization, regularization, and per-epoch validation. Each configuration is handled independently to ensure integrity.

regularization via an L1 term, gradient clipping for stability, and dynamic learning rate adjustment using `ReduceLROnPlateau`. During each epoch, model performance is evaluated on a dedicated validation set, and training proceeds with early stopping based on validation F1-score.

This setup guarantees fairness and comparability between training regimes and supports robust convergence monitoring. The pipeline is optimized for reproducibility, with consistent random seed control, fixed test splits, and no data leakage between training and evaluation.

4.1.4 Evaluation Protocol

To comprehensively assess the effectiveness of the VISION framework, a diverse set of evaluation metrics is employed, covering both conventional predictive performance and explanation-oriented criteria. These metrics are grouped into four categories:

Conventional Metrics

Standard classification metrics are used to evaluate predictive performance on the fixed, balanced test set:

- **Accuracy**: The proportion of correctly classified samples out of the total number of examples. While high accuracy is desirable, it can be misleading in imbalanced datasets—hence its interpretation is supported by other metrics.
- **Precision**: The proportion of predicted vulnerable samples that are actually vulnerable. A high precision indicates that the model avoids false positives (i.e., it does not incorrectly label benign code as vulnerable).
- **Recall**: The proportion of actual vulnerable samples that are correctly predicted. A high recall indicates that the model detects most of the truly vulnerable cases, avoiding false negatives.
- **F1-Score**: The harmonic mean of precision and recall. This metric balances both aspects, providing a robust summary of classification quality. High F1-scores signal strong, reliable performance, especially important in security-critical tasks where both false positives and false negatives are problematic.

Robustness Metrics

These metrics evaluate how well the model generalizes to challenging examples and avoids overfitting to spurious or superficial patterns.

- **Pair-Wise Accuracy (PWA)**: This metric quantifies how effectively the model distinguishes between original functions and their minimally modified counterfactual counterparts:
  - P-C (Pair-Correct): Percentage of pairs where the model correctly classifies both original and counterfactual samples with opposite labels. Higher is better, indicating good sensitivity to subtle but meaningful code changes.
  - P-V (Pair-Vulnerable): Percentage of pairs where both samples are incorrectly classified as vulnerable. Lower is better, as it suggests overgeneralization or excessive pessimism.

  - P-B (Pair-Benign): Percentage where both are wrongly predicted benign. Also lower is better, as it indicates the model may overlook critical security flaws.
  - P-R (Pair-Reversed): Percentage where the original and counterfactual predictions are flipped compared to their true labels. Lower values indicate more reliable decision boundaries.

- **Worst-Group Accuracy (WGA)**: Measures the classification performance on the most challenging subgroup within the data. Subgroups are automatically defined via K-means clustering over learned embeddings, intersected with class labels. A higher WGA reflects better generalization and reduced vulnerability to rare or structurally unique patterns. It is especially important for identifying brittle models that perform poorly on underrepresented but critical edge cases.

- **Neighborhood Purity**: Evaluates the clustering quality of the learned embedding space. For each function, its k-nearest neighbors (in latent space) are identified, and purity is measured as the proportion of neighbors with the same class. Higher scores imply stronger semantic coherence in the representation space, indicating that the model has learned to organize functions based on true vulnerability semantics rather than coincidental structural similarities.

Explanation Metrics

To assess how and why the model makes its predictions, a set of attribution-based metrics is used:

- **Intra-Class Attribution Variance**: Measures how consistent the explanation patterns are across different samples within the same class. It is computed as the variance of node-level importance scores among samples labeled either benign or vulnerable. Lower variance is desirable—it suggests that the model applies similar reasoning across similar functions, indicating more stable and generalizable interpretability.

- **Inter-Class Attribution Distance**: Measures the average distance between the explanation vectors of benign and vulnerable samples. Higher values indicate that the model distinguishes between the classes using clearly different reasoning patterns, which is a sign of good class separability in terms of explanatory behavior.

- **Node Score Dependency (Proposed)**: A new metric introduced in this work to assess inter-node influence in explanation scores. For each node, it measures how masking that node affects the importance of the others, yielding a matrix of node interdependencies. This allows detection of spurious dependencies—e.g., when unimportant context nodes unduly affect the model's reasoning. Semantically meaningful influence patterns support robust and interpretable modeling.

Visualization-Aided Analysis

Beyond quantitative metrics, model behavior is inspected qualitatively using the visualization module introduced in section 3.6. Visual comparisons of attribution maps, node graphs, and subgraph-based explanations allow direct observation of the model's attention and help verify whether its reasoning aligns with security-relevant code structures. This mode of evaluation is particularly useful in diagnosing failure cases and validating improvements introduced by counterfactual augmentation.

4.2 Performance Across Benchmarks

This section presents the classification performance of the detection model across different benchmark configurations composed of varying proportions of original and counterfactual code examples. Each configuration is trained independently using the setup described in section 4.1, and all are evaluated against the same fixed, balanced test set to ensure a fair and consistent comparison.

4.2.1 Analysis of Standard Metrics

To establish a foundational understanding of model behavior across training configurations, the conventional classification metrics—**accuracy**, **precision**, recall, and **F1-score**—are evaluated using a fixed, balanced test set. These metrics provide insights into the raw predictive capabilities of the model trained with different combinations of original and counterfactual data, as presented in Table 4.2.

Table 4.2: Performance of standard metrics (Accuracy, Precision, Recall, and F1-score) across original/counterfactual training splits on the balanced test.

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

The results reveal a clear trend: models trained solely on original examples (100/0) suffer from **extreme imbalance** in their predictive behavior. While the 100/0 configuration yields perfect precision (1.000), it performs poorly in terms of recall (0.036) and F1-score (0.069). This indicates that the model has learned to identify benign code with high confidence but fails to detect vulnerable cases, a hallmark of overfitting to superficial patterns common in original data.

Introducing even a small proportion of counterfactual data (e.g., 10% in the 90/10 split) substantially improves the model's generalization. Precision remains high (0.996), while recall increases to 0.737—yielding a significantly improved F1-score of 0.847. This early gain suggests that counterfactuals effectively sharpen the model's discriminatory ability.

The strongest overall results are observed in the range of balanced benchmarks—specifically, those with roughly equal proportions of original and counterfactual data (e.g., 50/50, 60/40). In this region, the model achieves its highest **F1-score (0.979)** and **recall (0.997)**, while maintaining a competitive **accuracy of 0.978**. These outcomes reflect a model that effectively captures the semantic nuances of both benign and vulnerable code without bias, particularly enhancing the detection of hard-to-classify vulnerable samples.

Additionally, **precision** reaches a peak of **0.998** for the 40/60 split, indicating that incorporating counterfactuals does not increase false positives. However, beyond this optimal range—especially when training is skewed heavily toward synthetic counterfactuals—the model begins to suffer. For instance, the 0/100 benchmark exhibits a substantial drop in **precision (0.726)** and **accuracy (0.799)**, suggesting a loss of grounding in authentic patterns and an increased tendency to overgeneralize.

These results highlight a key trade-off: while counterfactual data can significantly improve generalization and predictive balance, over-reliance on synthetic examples introduces performance degradation. Taken together, the analysis supports that balanced benchmarks (e.g., 50/50, 60/40) offer the most reliable performance, mitigating spurious correlations while preserving high-quality vulnerability detection.

4.2.2 Overfitting, Generalization, and Learning Stability

Beyond static test set metrics, dynamic training behavior provides critical insights into model generalization and overfitting tendencies. This section investigates how training trajectories evolve under different benchmark configurations, focusing on two representative cases: a balanced configuration (e.g., 50/50) and an extreme synthetic-heavy setting (0/100).

Figure 4.5 compares the training and validation accuracy curves of both configurations over training epochs. Each model is saved at the epoch where it reaches the highest validation F1-score, following the early stopping strategy outlined in section 4.1.

On the left, the balanced configuration demonstrates **stable and synchronized convergence**. Training and validation accuracy curves closely track one another, with minimal divergence throughout training. This indicates strong generalization, where the model learns patterns that translate well to unseen data. The smooth convergence suggests that counterfactual and original examples complement each other, enhancing the model's ability to capture robust decision boundaries.

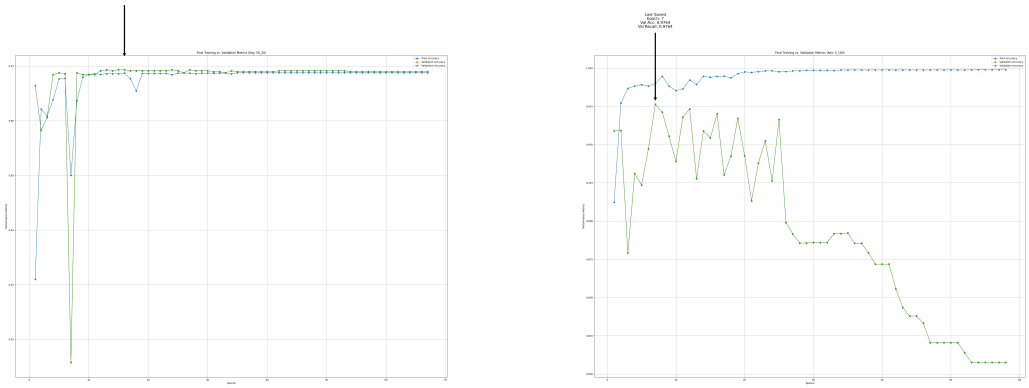


Figure 4.5: Comparison of training and validation accuracy curves for two benchmark configurations: (Left) Balanced training (50/50), and (Right) Fully synthetic training (0/100). While the 50/50 model maintains stable and converging curves, the 0/100 model overfits rapidly and exhibits degrading validation performance over time.

In contrast, the right-hand plot shows the synthetic-only configuration (0/100). Here, the model rapidly achieves near-perfect training accuracy, suggesting strong memorization. However, the validation accuracy curve shows significant fluctuations and a noticeable **degradation trend** as training progresses. This instability is a clear symptom of overfitting: the model learns to exploit superficial artifacts in the synthetic data that do not generalize to the more diverse test set. While early stopping helps prevent the most severe deterioration, the gap between training and validation accuracy highlights a brittleness in learning, rooted in the absence of authentic examples.

These observations affirm that a moderated counterfactual integration strategy—as seen in the balanced benchmarks—not only improves final evaluation metrics (as shown in Table 4.2) but also contributes to more stable, generalizable, and efficient training dynamics. The inclusion of real examples appears essential for maintaining grounding and avoiding shortcut learning, while the presence of counterfactuals encourages better semantic differentiation.

4.2.3 Optimal Performance Identification

After analyzing individual performance metrics and training behaviors across all augmentation benchmarks, this section aims to identify which configuration yields the most favorable balance between detection capability and model stability. The goal is not simply to maximize a single metric, but to select a configuration that maintains strong generalization, minimizes overfitting, and achieves balanced classification across benign and vulnerable code.

To facilitate this comparison, Figure 4.6 presents a scatter plot of test set accuracy versus recall for all benchmark configurations. Each point represents a model trained with a different original-to-counterfactual ratio. The figure captures the trade-off

between these two critical dimensions of performance.

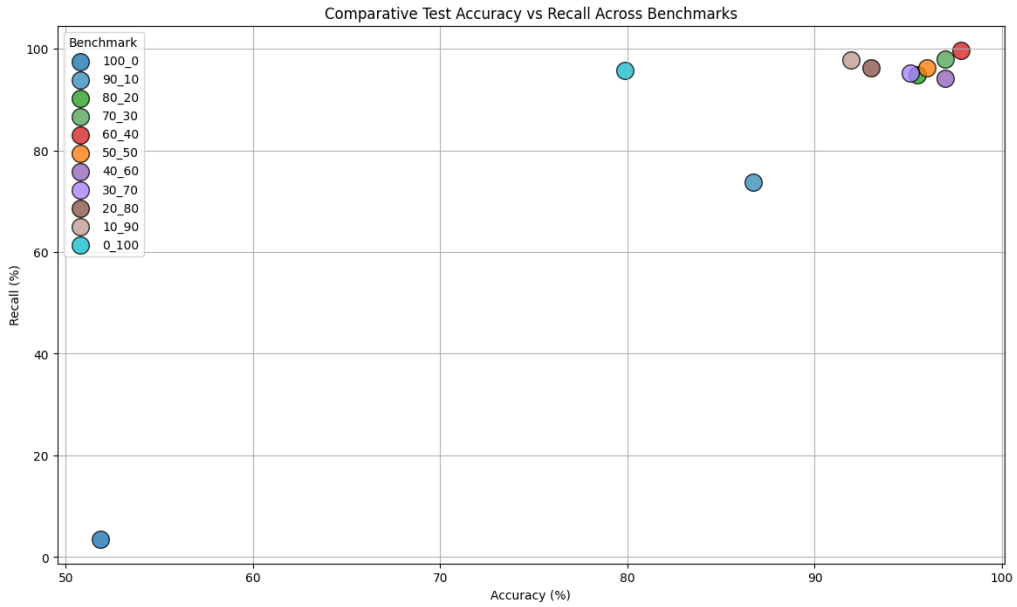


Figure 4.6: Comparative scatter plot of accuracy (x-axis) vs. recall (y-axis) on the test set for all original/counterfactual training splits. Balanced configurations (e.g., 50/50, 60/40) achieve both high accuracy and recall, while extreme configurations (100/0 and 0/100) suffer from overfitting or reduced precision.

On the left side of the plot, the 100/0 configuration is clearly isolated. It achieves perfect precision, but its recall is drastically limited—confirming its tendency to overfit to superficial patterns found in the original data. Conversely, the 0/100 configuration achieves very high recall (0.957), suggesting that it is extremely sensitive to vulnerability patterns. However, this comes at the cost of lower precision (0.726) and reduced accuracy (0.799), highlighting the risk of false positives and diminished reliability.

The most compelling region of the plot lies in the top-right quadrant, where multiple configurations cluster with both high accuracy (above 0.95) and recall (above 0.94). Among these, balanced configurations (e.g., 50/50 and 60/40) consistently occupy the frontier of optimality. The 60/40 configuration achieves the best F1-score (0.979), while the 50/50 model reaches the highest recall–accuracy tradeoff balance. These configurations strike the ideal compromise: the model is neither overly conservative (as in 100/0), nor excessively sensitive (as in 0/100), but demonstrates discriminative power grounded in semantically relevant features.

In practical terms, this reinforces a key insight: **counterfactuals**, when used in moderation, provide sufficient variability to **reduce shortcut learning** and **foster generalization**—without overwhelming the training signal with synthetic artifacts. It is this diversity and subtle contrast that ultimately leads to optimal decision boundaries.

4.3 Robustness and Spurious Correlation Analysis

Beyond predictive performance, the reliability of a vulnerability detection model hinges on its robustness to subtle variations and its ability to avoid **spurious correlations**. In the context of source code, spurious learning often occurs when a model associates vulnerability labels with superficial or dataset-specific patterns—such as fixed tokens or benign code idioms—rather than core semantic signals that indicate real security flaws. This section investigates how counterfactual-based augmentation influences model behavior under such conditions, through a suite of dedicated robustness and interpretability diagnostics.

To this end, a comprehensive set of metrics is examined, including **pair-wise accuracy**, which evaluates sensitivity to minor yet label-reversing changes; **worst-group accuracy**, which measures subgroup-level performance; and **neighborhood purity**, reflecting the class consistency in latent representation space. In addition, **attribution-based diagnostics** are explored, including **intra-class variance**, **inter-class attribution distance**, and the newly introduced **node score dependency metric**. These indicators provide deeper insight into how models reason, how consistent their explanations are, and how resilient they remain to structural perturbations or shortcut behaviors.

The results are presented in Table 4.3 and discussed across the following subsections, focusing on how different levels of counterfactual integration affect model robustness, generalization, and attribution alignment.

Table 4.3: A comprehensive evaluation is conducted across various training splits, assessing model 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 (reversed predictions). Higher P-C and lower P-V/P-B/P-R scores reflect stronger discriminatory ability. Worst-Group Accuracy (WGA, k=2–7) captures robustness across subgroups, with higher values indicating better performance under distributional shifts. Neighborhood Purity measures class consistency in the learned embedding space—higher values imply stronger semantic separation. Attribution-based metrics include Intra-class Attribution Variance (lower is better, indicating explanation consistency within classes) and Inter-class Attribution Distance (higher is better, showing clear distinction between classes).

Split	Acc	Prec	Rec	F1	P-C	P-V	P-B	P-R	WGA4	Purity	Intra-B	Intra-V	Inter-D
100/0	0.518	1.000	0.036	0.069	4.50	0.00	95.43	0.07	0.0073	0.707	0.01103	0.01027	0.00061
90/10	0.867	0.996	0.737	0.847	74.09	1.38	23.88	0.65	0.7052	0.907	0.01120	0.01035	0.00073
80/20	0.955	0.960	0.948	0.954	91.07	5.44	3.27	0.22	0.8757	0.953	0.01096	0.01046	0.00027
70/30	0.970	0.960	0.980	0.970	94.63	4.86	0.36	0.15	**0.8757**	0.962	0.01109	0.00995	0.00010
60/40	**0.978**	0.961	**0.997**	**0.979**	93.69	6.31	0.00	**0.00**	0.8703	**0.967**	0.01134	0.01030	0.00010
50/50	0.960	0.957	0.962	0.960	**95.79**	**0.44**	**0.00**	3.77	0.8555	0.944	**0.01061**	**0.01030**	**0.00160**
40/60	0.970	**0.998**	0.941	0.969	94.12	1.02	4.50	0.36	0.8092	0.966	0.01122	0.01036	0.00017
30/70	0.951	0.949	0.953	0.951	87.52	8.13	3.85	0.51	0.8266	0.941	0.01101	0.01010	0.00038
20/80	0.930	0.904	0.962	0.932	70.97	27.72	1.02	0.29	0.8497	0.929	0.01144	0.01036	0.00028
10/90	0.919	0.875	0.978	0.924	77.94	20.54	0.65	0.87	0.8152	0.910	0.01103	0.01046	0.00008
0/100	0.799	0.726	0.957	0.826	41.51	57.40	0.65	0.44	0.5030	0.856	0.01122	0.01007	0.00099

4.3.1 Spuriousness in Source Code

In machine learning for code vulnerability detection, **spurious correlations** arise when models learn to rely on superficial patterns that are statistically—but not semantically—correlated with vulnerability labels. These misleading patterns are particularly common in code due to repetitive syntax, reused variables, and standard idioms that appear across both benign and vulnerable examples. When such features are mistakenly learned as discriminative signals, models fail to generalize and instead rely on dataset-specific shortcuts.

The CWE-20 vulnerability class, which concerns improper input validation, is especially susceptible to this issue. For example, a model may associate the presence of certain variable names or safe usage patterns with benign behavior, without understanding the origin or security context of those variables. This is dangerous in practice, as the same function may be safe or unsafe depending on where inputs originate and how they are validated.

Figure 4.7 demonstrates this phenomenon with a concrete code example. The top listing shows a benign implementation of `param_get_mode`, where the mode variable is assigned internally from a trusted source (`aa_g_profile_mode`). This usage is safe and complies with CWE-20, as there is no reliance on unvalidated external input. However, a model exposed only to this structure might erroneously learn that any appearance of mode in the `sprintf()` call is safe.

Original benign code: No CWE-20 issue.

```
1 static int param_get_mode(char *buffer, struct kernel_param *kp)
2 {
3     if (!capable(CAP_MAC_ADMIN))
4         return -EPERM;
5     if (!apparmor_enabled)
6         return -EINVAL;
7     int mode = aa_g_profile_mode; // Potentially spurious statement
8     return sprintf(buffer, "\%s", profile\_mode\_names[mode]);
9 }
```

Vulnerable counterexample: Unvalidated `mode` input introducing a CWE-20 flaw

```
1 static int param_get_mode(char *buffer, struct kernel_param *kp, int mode)
2 {
3     if (!capable(CAP_MAC_ADMIN))
4         return -EPERM;
5     if (!apparmor_enabled)
6         return -EINVAL;
7     // Introduce a vulnerability by not checking the bounds of mode
8     return sprintf(buffer, "%s", profile_mode_names[mode]);
9 }
```

Figure 4.7: 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.

The bottom listing introduces a minimal change that flips the label to vulnerable: the `mode` variable is now passed as an argument to the function—meaning it may come from an external, potentially untrusted source—and is directly used

in a memory access operation without bounds checking. While this change is semantically critical from a security standpoint, it is syntactically subtle. Without sufficient contrastive data, a model may still associate `mode` with benign behavior and fail to flag this version as vulnerable.

This example illustrates how models can learn spurious associations unless exposed to carefully controlled counterfactuals. Training on both versions enables the model to distinguish semantic intent, rather than memorizing token-level patterns. The VISION framework leverages this principle by systematically generating such counterfactual pairs to minimize shortcut learning.

4.3.2 Pair-Wise Accuracy (PWA)

Pair-wise accuracy, introduced in [Din+24], is a diagnostic metric specifically tailored to evaluate model sensitivity to minimal but semantically meaningful changes in source code. It is particularly relevant when analyzing models trained on datasets augmented with counterfactual examples, as it measures the model's ability to distinguish between nearly identical functions that differ only in their vulnerability label.

Each pair consists of two functions: the **original** (either benign or vulnerable) and its corresponding **counterfactual** (whose label is flipped through a minimally invasive edit). Ideally, a robust model should correctly classify both functions, attributing the change in prediction to the introduced semantic modification rather than superficial features. In contrast, a model affected by shortcut learning or spurious correlations may fail to detect the meaningful distinction.

The pair-wise metric decomposes into four components:

- **P-C (Correct Contrast)**: Proportion of pairs for which one function is correctly classified as vulnerable and the other as benign, matching their respective ground-truth labels. *Higher is better*, as it reflects accurate differentiation between subtle label-flipping edits.
- **P-V (Both Vulnerable)**: Proportion of pairs where both functions are incorrectly predicted as vulnerable. *Lower is better*, since this indicates the model is not overgeneralizing vulnerability cues.
- **P-B (Both Benign)**: Proportion of pairs where both functions are incorrectly predicted as benign. *Lower is better*, as high values suggest the model is missing critical vulnerability indicators.
- **P-R (Reversed)**: Proportion of pairs where both predictions are flipped with respect to their ground-truth labels. *Lower is better*, because it highlights prediction inconsistency and unreliable learning behavior.

A high P-C and low values for P-V, P-B, and P-R indicate effective contrastive understanding and low susceptibility to spurious patterns.

Evaluation Results

Table 4.4 reports the pair-wise classification outcomes across all benchmark configurations. The highest P-C value is observed for the **50/50 benchmark**, with **95.79%** of all pairs correctly classified in contrast to one another. This result underscores the effectiveness of balanced augmentation in encouraging semantic sensitivity.

On the other hand, extreme splits such as 100/0 and 0/100 show clear signs of poor contrastive learning. The 100/0 model, trained exclusively on original examples, achieves a P-C of only 4.50% and mistakenly predicts both functions as benign in over 95% of cases (P-B = 95.43%). Similarly, the 0/100 model incorrectly predicts both examples as vulnerable 57.4% of the time (P-V = 57.40%). These patterns suggest reliance on dataset-specific artifacts rather than true semantic understanding.

The results further show that balanced counterfactual inclusion (e.g., 50/50) consistently improves P-C values and suppresses P-V/P-B confusion. Notably, P-R scores remain low in most cases, indicating that label reversal is rare, and misclassifications tend to cluster in one direction.

Table 4.4: Pair-wise classification breakdown for each original/counterfactual split. High P-C and low P-V, P-B, and P-R indicate successful semantic discrimination between original and counterfactual examples.

Split	**P-C** (%)	**P-V** (%)	**P-B** (%)	**P-R** (%)
100/0	4.50	0.00	95.43	0.07
90/10	74.09	1.38	23.88	0.65
80/20	91.07	5.44	3.27	0.22
70/30	94.63	4.86	0.36	0.15
60/40	93.69	6.31	0.00	**0.00**
50/50	**95.79**	**0.44**	**0.00**	3.77
40/60	94.12	1.02	4.50	0.36
30/70	87.52	8.13	3.85	0.51
20/80	70.97	27.72	1.02	0.29
10/90	77.94	20.54	0.65	0.87
0/100	41.51	57.40	0.65	0.44

4.3.3 Worst-Group Accuracy (WGA)

Robust models should maintain performance not only on average, but also across all types of inputs—including the most challenging or underrepresented subgroups. **Worst-Group Accuracy (WGA)** is a robustness metric designed to capture the weakest point of a model's performance across such latent subpopulations. It evaluates whether the model generalizes well across diverse structural and semantic patterns, and whether its learning is biased toward dominant data modes [Idr+22].

Since explicit spurious or confounding attributes are not available in this setting, WGA is computed using an unsupervised strategy. After training, code embeddings are extracted for all test samples, and K-means clustering is applied to partition the data into latent groups based on learned structure. Each group is defined as the intersection of a cluster ID and a ground-truth class label (benign or vulnerable). Small groups with fewer than 1% of the total dataset are discarded to ensure statistical stability. WGA is then defined as the minimum classification accuracy over the remaining groups.

Formally, given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ and a trained model f, latent groups are defined 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 its true label y_i.

For each group $g \in \mathcal{G}$ such that $|g| > 0.01N$, the accuracy is computed as:

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

Then, the Worst-Group Accuracy (WGA) is defined as:

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

Higher values of WGA indicate stronger worst-case performance, suggesting that the model avoids overfitting to specific patterns or relying on brittle heuristics. Conversely, a low WGA value implies that at least one group is poorly served by the model, likely due to spurious correlations or insufficient training diversity.

Evaluation Results

As reported in Table 4.3, the model trained solely on original examples (100/0) exhibits extremely poor performance in worst-group accuracy (WGA), with values remaining below 2% across all clustering granularities ($k = 2$ to $k = 7$) This indicates a high degree of overfitting to superficial dataset artifacts, leading to brittleness when generalizing to structurally diverse subgroups.

In contrast, models trained using balanced splits—such as 60/40 and 50/50—demonstrate a more stable WGA profile across different values of k While their top scores might not always be the highest for individual k settings, the decrease in WGA from $k = 2$ to $k = 7$ is significantly more moderate compared to other configurations. This consistency suggests that balanced data composition fosters more uniform performance across latent subgroups, reducing susceptibility to hidden biases and dataset shortcuts.

Meanwhile, benchmarks heavily skewed toward synthetic examples (e.g., 0/100) again show degraded WGA, with all values falling below 0.54. This reinforces that while counterfactuals enhance robustness, excessive reliance on them may lead to incomplete generalization due to reduced exposure to real-world variability.

Table 4.5: Worst-Group Accuracy (WGA) across training benchmarks for different clustering granularities $k = 2$ to $k = 7$. Higher values indicate stronger robustness across latent subgroups.

Split	WGA2	WGA3	WGA4	WGA5	WGA6	WGA7
100/0	0.0171	0.0096	0.0073	0.0067	0.0058	0.0067
90/10	0.7309	0.7156	0.7052	0.7126	0.5909	0.5952
80/20	**0.9115**	**0.8828**	**0.8757**	0.8512	0.8444	0.8205
70/30	0.9056	0.8745	0.8757	**0.8595**	0.8444	0.8205
60/40	0.9056	0.8745	0.8703	0.8512	**0.8444**	0.8205
50/50	0.8991	0.8667	0.8555	0.8087	0.7955	0.8095
40/60	0.8471	0.8089	0.8092	0.7739	0.7955	0.8067
30/70	0.8777	0.8400	0.8266	0.7739	0.7727	0.7857
20/80	0.8820	0.8622	0.8497	0.8174	0.8182	**0.8333**
10/90	0.8584	0.8350	0.8152	0.8265	0.8149	0.8099
0/100	0.5398	0.4983	0.5030	0.4966	0.4754	0.4742

The WGA analysis highlights the importance of **balanced augmentation strategies**. Models trained with a proportionate mix of original and counterfactual data achieve more reliable and **subgroup-fair** predictions, exhibiting resilience to overfitting and shortcut learning across varying structural complexities.

4.3.4 Neighborhood Analysis in Embedding Space

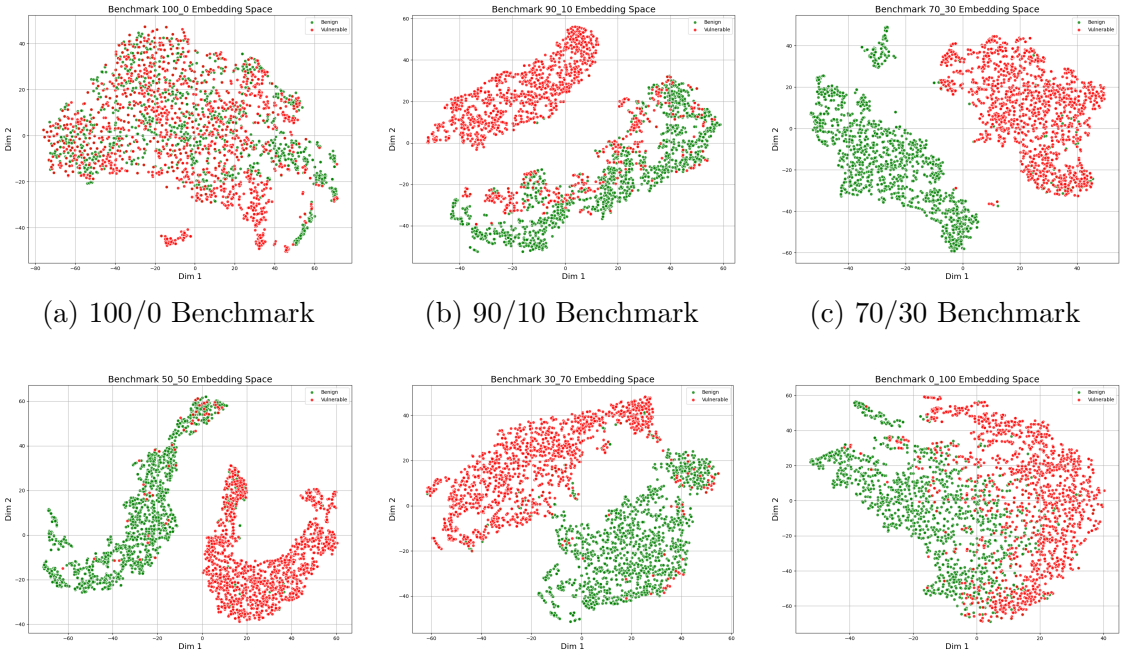
One of the key goals of counterfactual augmentation is to steer the model toward learning semantically meaningful representations. To evaluate this, graph-level embeddings are visualized using t-SNE projections, allowing qualitative comparison of the learned **latent space** under different training splits. Six benchmark configurations are presented in Figure 4.8, ranging from fully original to fully synthetic, including several intermediate configurations with varying degrees of balance.

Evaluation Results

As visible in Figure 4.8, the fully original (100/0) and fully synthetic (0/100) models show the least structured class separation. Embeddings for benign and vulnerable samples are heavily entangled, suggesting that the model has failed to internalize meaningful vulnerability signals. This aligns with earlier performance results, where these extreme configurations suffered from overfitting or underfitting.

In contrast, **well-balanced training sets** (e.g., 50/50, 70/30) result in far more **structured latent spaces**. Classes form distinct, separable clusters, indicating that the model has learned to represent benign and vulnerable samples with clear semantic differences. Intermediate splits such as 30/70 and 90/10 show partial structure, with varying degrees of cluster overlap.

These visual trends are quantitatively supported by the neighborhood purity values summarized in Table 4.6. Higher **purity scores** reflect stronger class consistency in



(a) 100/0 Benchmark (b) 90/10 Benchmark (c) 70/30 Benchmark

(d) 50/50 Benchmark (e) 30/70 Benchmark (f) 0/100 Benchmark

Figure 4.8: t-SNE projections of graph-level embeddings from models trained on different benchmarks. Green dots represent benign samples and red dots denote vulnerable samples. Balanced training configurations (e.g., 50/50, 70/30) produce clearer separation between classes, while extremes like 100/0 and 0/100 result in overlapping clusters and less structured embedding spaces.

the learned embedding space. Balanced configurations such as 60/40 and 70/30 yield the most consistent clustering, confirming that counterfactual augmentation leads to more semantically aligned latent representations. In contrast, extreme settings such as 100/0 and 0/100 result in lower purity scores, highlighting the role of balance in promoting structured embedding separation.

Table 4.6: Neighborhood Purity scores for each benchmark configuration. Higher values indicate better class-consistent clustering in the embedding space. Balanced configurations (e.g., 60/40, 50/50, 40/60) achieve the highest purity.

Split	**Neighborhood Purity**
100/0	0.707
90/10	0.907
80/20	0.953
70/30	0.962
60/40	**0.967**
50/50	0.944
40/60	0.966
30/70	0.941
20/80	0.929
10/90	0.910
0/100	0.856

4.3.5 Intra-Class Attribution Variance and Inter-Class Attribution Distance

While traditional evaluation metrics assess a model's predictive performance, they offer limited insight into how a model arrives at its decisions. To address this gap, two attribution-based metrics are introduced: **Intra-Class Attribution Variance** and **Inter-Class Attribution Distance**. These are derived from node-level explanation vectors produced by the Illuminati explainer and provide a deeper view into the consistency and semantic structure of learned model reasoning.

Intra-Class Attribution Variance

This metric quantifies how consistent a model's explanations are across examples belonging to the **same class**. For each class (benign and vulnerable), attribution vectors are collected from all samples, and their statistical variance is computed. A lower variance value suggests that the model attributes its predictions to similar structural patterns across different examples—an indicator of stable, generalizable reasoning. Conversely, high intra-class variance may reflect reliance on spurious, input-specific signals that do not generalize well.

Lower values are desirable. They indicate that the model is internally coherent and bases decisions on consistent, semantically grounded cues rather than superficial or sample-specific artifacts.

Inter-Class Attribution Distance

This metric evaluates how different the average attribution patterns are **between classes** (benign and vulnerable). The mean attribution vector for each class is computed, and the Euclidean distance between these vectors is measured. A higher inter-class distance suggests that the model distinguishes vulnerabilities using clearly different structural cues, rather than blurring class boundaries.

Higher values are desirable. A large separation between classes indicates that the model captures distinctive vulnerability signatures rather than learning ambiguous or overlapping representations.

Evaluation Results

The results for both metrics are presented in Table 4.7. Intra-class attribution variance remains relatively low across all benchmarks, with little fluctuation, indicating that the model tends to apply consistent reasoning within each class regardless of the data composition.

However, the inter-class attribution distance varies more noticeably and peaks at the **balanced 50/50 configuration** (0.00160), which demonstrates the highest semantic separation between benign and vulnerable explanations. This reinforces earlier findings that balanced augmentation not only improves prediction quality but also enhances the **clarity** and **reliability** of model interpretations.

In contrast, extreme splits—such as 100/0 or 0/100—exhibit reduced class separation, likely due to overfitting (in the case of original-only data) or lack of semantic grounding (in the case of synthetic-only data).

Table 4.7: Attribution-based explanation metrics for each benchmark. Intra-Class Variance is computed separately for benign (Intra-Class Ben. Var.) and vulnerable (Intra-Class Vul. Var.) classes. Inter-Class Attribution Distance (Inter-Class Dist.) reflects how distinct the average attribution patterns are between classes. Lower intra-class variance and higher inter-class distance are desirable.

Split	Intra-Class Ben. Var.	Intra-Class Vul. Var.	Inter-Class Dist.
100/0	0.01103	0.01027	0.00061
90/10	0.01120	0.01035	0.00073
80/20	0.01096	0.01046	0.00027
70/30	0.01109	0.00995	0.00010
60/40	0.01134	0.01030	0.00010
50/50	**0.01061**	**0.01030**	**0.00160**
40/60	0.01122	0.01036	0.00017
30/70	0.01101	0.01010	0.00038
20/80	0.01144	0.01036	0.00028
10/90	0.01103	0.01046	0.00008
0/100	0.01122	0.01007	0.00099

4.3.6 Node Score Dependency

To complement the robustness and interpretability evaluation metrics, this work introduces **Node Score Dependency**, a novel analysis designed to reveal the relational structure of attributions in graph-based vulnerability detection. Unlike conventional attribution methods that assess node importance in isolation, this metric captures how the perceived relevance of one node is affected by the presence or absence of others. It provides a deeper view into the **interaction dynamics between graph components** and exposes potential shortcut behaviors or structural fragility within the model's explanation process.

Formally, for a graph composed of n nodes, let $\text{score}_j^{\text{orig}}$ denote the attribution score of node j in the unaltered graph. For each node i, a modified graph is created by masking or removing node i, and the attribution scores are recomputed using a post hoc explainer. The absolute difference between the original and modified scores of node j defines the dependency metric:

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

This yields a square matrix $M \in \mathbb{R}^{n \times n}$, where $M_{i,j}$ quantifies the influence of node i on the attribution of node j. Diagonal entries $M_{i,i}$ reflect each node's self-dependence, while off-diagonal values characterize cross-node attribution sensitivity.

```
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 the bounds of mode
    return sprintf(buffer, "%s", profile_mode_names[mode)]);
}
```

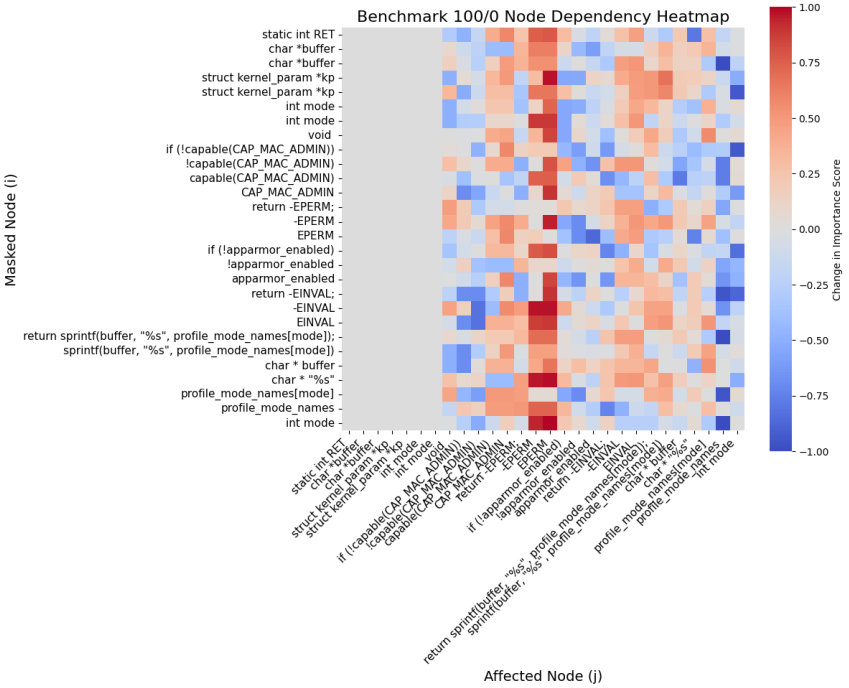
Figure 4.9: **Code example for Node Score Dependency Analysis.** This vulnerable function is used to evaluate how the removal of individual nodes affects the attribution scores of others. The unvalidated `mode` parameter introduces a CWE-20 flaw, making it suitable for analyzing attribution dependency and interaction patterns across code components.

To ground the attribution dynamics in a concrete context, this analysis focuses on the vulnerable counterexample introduced in Figure 4.9. This function illustrates a CWE-20 vulnerability where the variable `mode`—originating from an untrusted source—is directly passed to `sprintf()` without validation. It represents a typical scenario where spurious correlations may arise if the model learns to associate the mere presence of common variables (e.g., `mode`) with benign behavior, instead of identifying the security-critical misuse patterns. This example serves as the basis for evaluating attribution shifts and inter-node dependencies across training regimes using the proposed Node Score Dependency metric.

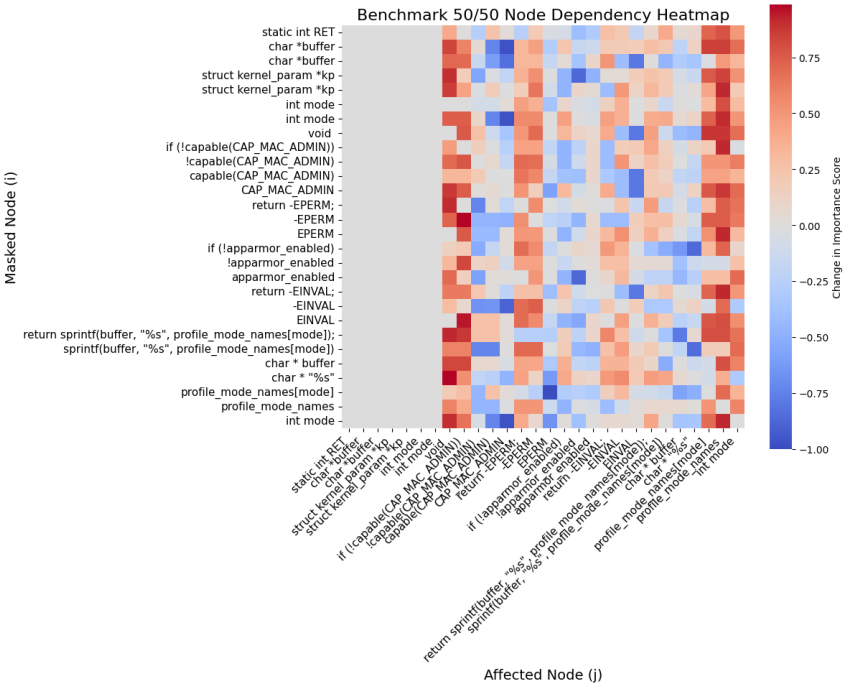
The heatmaps in Figure 4.10 illustrate this concept through a comparative analysis of two trained models on the same vulnerable example (see Figure 4.7)—one trained on 100% original data and the other on a balanced 50/50 mixture of original and counterfactual examples. Each cell's color intensity represents the magnitude of attribution change, with red indicating stronger influence and blue denoting weaker or null effect.

The 100/0 model shows a concentrated attribution dependency centered around the node corresponding to `EPERM`, a standard error return value. Despite its contextual significance in the example, this node is not directly indicative of a CWE-20 vulnerability. Its prominence in the dependency map suggests that the model may have latched onto superficial features, likely due to repeated occurrences in the training data. In contrast, semantically critical elements such as the user-controlled variable `mode` and its usage within `profile_mode_names[mode]` exhibit little to no influence, revealing a lack of semantic grounding.

Conversely, the dependency structure of the 50/50 model displays a more diffuse and context-aware interaction pattern. Nodes associated with improper input validation—such as mode and `profile_mode_names`—exert notable influence over other components in the graph. The dependency matrix highlights how attribution propagates through semantically relevant control and data flow pathways, indicating improved model comprehension of vulnerability-inducing structures. This alignment with control flow (e.g., if statements and return branches) supports the notion that balanced counterfactual augmentation encourages more faithful and generalizable reasoning.



(a) 100/0 Model Node Dependency Map



(b) 50/50 Model Node Dependency Map

Figure 4.10: Node Score Dependency heatmaps for the same vulnerable function under two training regimes. (a) 100/0 benchmark, where dependencies are narrowly concentrated on superficial nodes. (b) 50/50 benchmark, which exhibits semantically meaningful and distributed dependency patterns. Each cell shows how removing one node (rows) affects the attribution of another (columns). Red denotes strong influence; blue indicates minimal or no effect.

These findings highlight the utility of Node Score Dependency as an interpretability-aware diagnostic. By exposing how explanation signals propagate across graph components, this metric reveals the internal logic—or lack thereof—behind a model's predictions. When applied across training regimes, it confirms that a balanced integration of counterfactual examples fosters models that are both structurally robust and semantically aligned. As a result, Node Score Dependency offers a valuable lens into **explanation quality**, enabling researchers and practitioners to identify spurious dependencies, diagnose shortcut behaviors, and validate attribution coherence in GNN-based vulnerability detection systems.

4.3.7 Summary of Robustness and Spuriousness Findings

The analyses conducted in this section provide strong empirical and interpretive support for the hypothesis that **counterfactual data augmentation mitigates shortcut learning** and enhances model robustness in the context of source code vulnerability detection.

Across all evaluated robustness metrics, models trained with **balanced compositions** of original and counterfactual examples (e.g., 50/50 and 60/40) **consistently outperform extreme configurations**. Pair-wise accuracy reveals that these balanced models are better at distinguishing semantically subtle vulnerabilities, while avoiding prediction inconsistencies caused by superficial patterns. Worst-group accuracy further confirms this robustness, with balanced models maintaining higher and more stable performance across latent subgroups derived from unsupervised clustering.

In terms of **representation quality**, neighborhood purity results and t-SNE projections show that models trained with moderate counterfactual integration develop **more semantically structured** and class-consistent embedding spaces. These embeddings reflect a **better internal organization** of vulnerability patterns, rather than reliance on dataset-specific idiosyncrasies.

At the explanation level, attribution-based metrics—including intra-class attribution variance and inter-class attribution distance—demonstrate that balanced training leads to **more consistent** and **discriminative reasoning patterns**. The novel Node Score Dependency analysis offers additional interpretive depth, highlighting how counterfactual training helps the model internalize meaningful inter-node interactions and reduces reliance on spurious context nodes.

Collectively, these results underscore the effectiveness of the proposed counterfactual augmentation strategy in addressing key challenges in GNN-based vulnerability detection. By exposing models to semantically minimal but security-relevant variations, the approach not only **boosts predictive accuracy** but also fosters more **robust**, **generalizable**, and **interpretable learning behavior**.

Chapter 5

Conclusions

This final section synthesizes the key findings of the study, outlines the primary **contributions** made by the VISION framework, and reflects on the broader implications of the results. While the proposed approach demonstrates significant improvements in robustness, explanation quality, and generalization, it is not without limitations. Accordingly, this section also discusses the scope of the current work and presents several promising directions for future research.

5.1 Achievements and Summary of Contributions

This work introduced VISION, a framework for enhancing robustness and interpretability in source code vulnerability detection via counterfactual data augmentation. The primary objective was to mitigate spurious correlations and improve model generalization in Graph Neural Network (GNN)-based systems by incorporating semantically meaningful perturbations into the training process.

The key achievements and contributions of this work can be summarized as follows:

Counterfactual Data Augmentation Strategy

A **novel augmentation pipeline** was developed to generate minimally modified counterfactual examples that invert vulnerability labels while preserving semantic validity. This strategy successfully exposed models to subtle but critical variations in input structure, fostering learning that prioritizes semantically grounded vulnerability patterns over dataset-specific artifacts.

CWE-20-CFA Benchmark Suite

An extensive benchmark was constructed using **CWE-20 examples**, enabling systematic evaluation across varying degrees of counterfactual integration. The benchmark allows fine-grained analysis of how original and synthetic data compositions affect model behavior, robustness, and explanation quality.

Empirical Gains in Predictive and Robustness Metrics

Models trained with balanced compositions of original and counterfactual examples exhibited substantial improvements in standard metrics (accuracy, precision, recall,

F1), and advanced robustness diagnostics such as **Pair-Wise Accuracy** and **Worst-Group Accuracy (WGA)**. These results demonstrate the framework's ability to reduce overfitting and enhance reliability under diverse conditions.

Attribution Consistency and Explanation Quality

VISION not only improved predictive performance but also enhanced the **semantic structure** of explanations. This was evidenced by favorable scores in Intra-Class Attribution Variance, Inter-Class Attribution Distance, and Node Score Dependency—novel metrics introduced in this work to evaluate the fidelity and discriminability of model reasoning.

Visualization Module for Qualitative Analysis

An interactive, Gradio-based visualization interface was designed to inspect code-level predictions and explanations. The tool provides side-by-side comparison of original and counterfactual examples, attribution highlights on both code and graph views, and subgraph analysis modes, facilitating human-in-the-loop model interpretation and debugging.

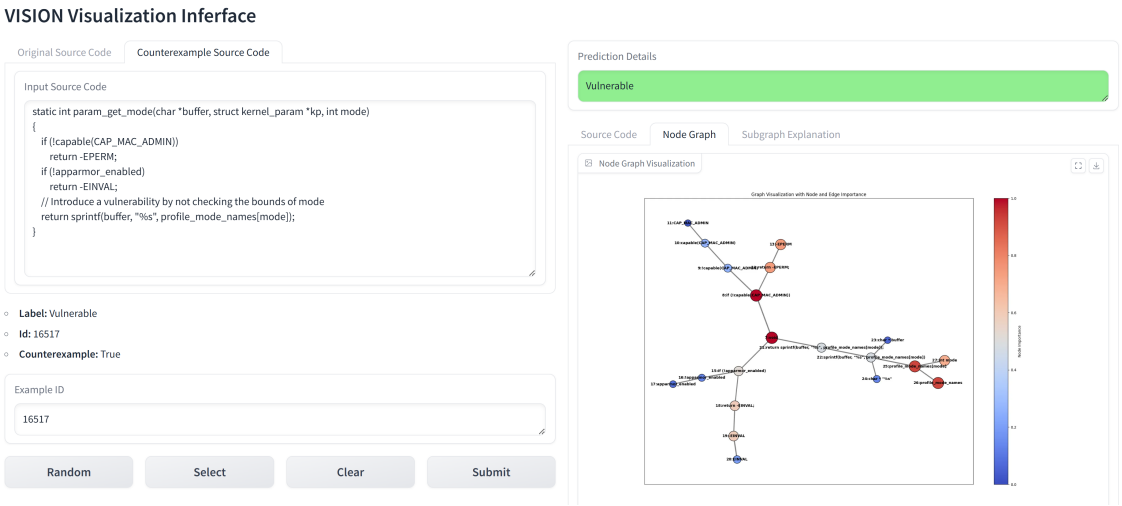


Figure 5.1: The visualization interface provides a human-in-the-loop environment for interpreting model decisions. It enables users to inspect predictions, highlight node-level attributions on both code and graph views, and analyze positive, negative, and optimal subgraphs. This interpretability support is critical for understanding model behavior and verifying explanation reliability in real-world scenarios.

Together, these contributions validate VISION as an effective, generalizable, and explainable approach to secure software analysis using Graph Neural Networks. The proposed framework not only improves predictive outcomes but also delivers deeper insight into model decision-making—crucial for the deployment of trustworthy vulnerability detection systems.

5.2 Limitations

While the proposed VISION framework demonstrates substantial improvements in robustness, generalization, and interpretability for vulnerability detection, several limitations must be acknowledged:

I. **Single Vulnerability Class Focus**: This work concentrates exclusively on CWE-20 (Improper Input Validation). Although this class is highly impactful and frequent in real-world applications, the conclusions drawn may not fully generalize to other vulnerability types such as buffer overflows or authorization issues. Further validation on a broader vulnerability spectrum is necessary to assess transferability.

II. **Manual Inspection of Synthetic Examples**: The framework relies on a large number of LLM-generated counterfactual examples—nearly 15,000 in total. Due to the scale, full manual verification is infeasible. A small sample of 50 examples was manually reviewed to validate generation quality. However, more rigorous, scalable verification techniques are required to ensure the semantic correctness and utility of all generated counterfactuals.

III. **Binary Classification Limitation**: The current version of VISION supports only binary classification (i.e., benign vs. vulnerable). Many real-world vulnerability scenarios involve multi-class or multi-label settings, such as detecting multiple CWE types within the same function. Extending the framework to these more complex classification tasks would significantly broaden its utility.

IV. **Model-Specific Implementation**: The framework is implemented using the Devign model and the Illuminati explainer. While these choices are justified by their relevance and performance, the extent to which VISION generalizes across other GNN architectures or explanation techniques remains untested.

V. **Visualization Scalability**: Although the visualization module is effective for qualitative inspection, its utility may degrade when applied to very large codebases or dense graph structures. Enhancing layout algorithms and interactive controls would be necessary for broader usability.

5.3 Future Work

Building on the current capabilities and limitations of the VISION framework, several promising directions can guide future research and development:

I. **Extension to Broader Vulnerability Classes**: Expanding the counterfactual data augmentation strategy to cover a wider range of CWE categories would allow testing the framework's robustness and generalizability across different types of software vulnerabilities. This includes more complex security flaws such as buffer overflows, injection vulnerabilities, or access control errors.

II. **Advanced Counterfactual Generation Techniques**: Future efforts may incorporate adversarial and probabilistic counterfactuals to diversify the training distribution further. These approaches could generate more challenging or boundary-pushing examples that increase model robustness and improve semantic discrimination.

III. **Automated Validation via Static or Dynamic Analysis Tools**: To reduce the reliance on manual inspection of LLM-generated examples, program analysis tools (e.g., static analyzers, symbolic execution engines) could be integrated into the generation pipeline. These tools would assist in automatically verifying that counterfactuals maintain syntax correctness and introduce meaningful security modifications.

IV. **Multi-class and Multi-label Vulnerability Classification**: Future work could generalize the framework to handle multi-class or multi-label vulnerability detection. This would reflect more realistic software security scenarios, where functions may exhibit multiple overlapping flaws or belong to distinct vulnerability families.

V. **Explanation-Guided Robust Training**: Incorporating explanation feedback into the training process may help prioritize semantically meaningful features during model optimization. This explanation-aware training approach could reduce shortcut learning further and encourage more human-aligned reasoning.

VI. **Visualization of Control Flow and Temporal Dependencies**: Expanding the visualization interface to support dynamic aspects—such as control-flow paths, taint propagation, or temporal execution traces—would enrich user understanding and support more comprehensive code audits and debugging.

Pursuing these future directions can significantly strengthen the technical foundation and practical utility of the VISION framework. Each proposal aligns with its core goals—enhancing robustness, reducing spuriousness, and improving interpretability. Expanding to broader vulnerability types, integrating validation tools, and incorporating explanation-guided training would make VISION more versatile and impactful. These advancements are essential steps toward building trustworthy, AI-assisted systems for secure software development.

Bibliography

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: https://doi.org/10.1145/24039.24041.

[Neu+07] Stephan Neuhaus et al. "Predicting vulnerable software components". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 529–540. ISBN: 9781595937032. DOI: 10.1145/1315245.1315311. URL: https://doi.org/10.1145/1315245.1315311.

[Sca+09] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[Yam+14] Fabian Yamaguchi et al. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. 2016. arXiv: 1602.04938 [cs.LG]. URL: https://arxiv.org/abs/1602.04938.

[LL17] Scott Lundberg and Su-In Lee. *A Unified Approach to Interpreting Model Predictions*. 2017. arXiv: 1705.07874 [cs.AI]. URL: https://arxiv.org/abs/1705.07874.

[CV18] Boris Chernis and Rakesh Verma. "Machine Learning Methods for Software Vulnerability Detection". In: *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*. IWSPA '18. Tempe, AZ, USA: Association for Computing Machinery, 2018, pp. 31–39. ISBN: 9781450356343. DOI: 10.1145/3180445.3180453. URL: https://doi.org/10.1145/3180445.3180453.

[Li+18] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: (Jan. 2018). DOI: 10.14722/ndss.2018.23158.

[WMR18] Sandra Wachter, Brent Mittelstadt, and Chris Russell. *Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR*. 2018. arXiv: 1711.00399 [cs.AI]. URL: https://arxiv.org/abs/1711.00399.

[Yin+19] Rex Ying et al. "GNNExplainer: Generating Explanations for Graph Neural Networks". In: *NeurIPS*. 2019.

[Zho+19] Yaqin Zhou et al. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. 2019. arXiv: 1909.03496 [cs.SE]. URL: https://arxiv.org/abs/1909.03496.

[Fen+20] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[KHL20] Divyansh Kaushik, Eduard Hovy, and Zachary C. Lipton. *Learning the Difference that Makes a Difference with Counterfactually-Augmented Data*. 2020. arXiv: 1909.12434 [cs.CL]. URL: https://arxiv.org/abs/1909.12434.

[Luo+20] Dongsheng Luo et al. *Parameterized Explainer for Graph Neural Network*. 2020. arXiv: 2011.04573 [cs.LG]. URL: https://arxiv.org/abs/2011.04573.

[Sag+20] Shiori Sagawa et al. *Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization*. 2020. arXiv: 1911.08731 [cs.LG]. URL: https://arxiv.org/abs/1911.08731.

[VT20] Minh N. Vu and My T. Thai. *PGM-Explainer: Probabilistic Graphical Model Explanations for Graph Neural Networks*. 2020. arXiv: 2010.05788 [cs.LG]. URL: https://arxiv.org/abs/2010.05788.

[ZZ20] Xiang Zhang and Marinka Zitnik. *GNNGuard: Defending Graph Neural Networks against Adversarial Attacks*. 2020. arXiv: 2006.08149 [cs.LG]. URL: https://arxiv.org/abs/2006.08149.

[Ahm+21] Wasi Uddin Ahmad et al. *Unified Pre-training for Program Understanding and Generation*. 2021. arXiv: 2103.06333 [cs.CL]. URL: https://arxiv.org/abs/2103.06333.

[Guo+21] Daya Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. 2021. arXiv: 2009.08366 [cs.SE]. URL: https://arxiv.org/abs/2009.08366.

[LWN21] Yi Li, Shaohua Wang, and Tien N. Nguyen. "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*. ACM, Aug. 2021, pp. 292–303. DOI: 10.1145/3468264.3468597. URL: http://dx.doi.org/10.1145/3468264.3468597.

[RMP21] Alexis Ross, Ana Marasović, and Matthew E. Peters. *Explaining NLP Models via Minimal Contrastive Editing (MiCE)*. 2021. arXiv: 2012.13985 [cs.CL]. URL: https://arxiv.org/abs/2012.13985.

[WP21] Lilapati Waikhom and Ripon Patgiri. *Graph Neural Networks: Methods, Applications, and Opportunities*. 2021. arXiv: 2108.10733 [cs.LG]. URL: https://arxiv.org/abs/2108.10733.

[Wan+21] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL]. URL: https://arxiv.org/abs/2109.00859.

[ZW21] Noah Ziems and Shaoen Wu. *Security Vulnerability Detection Using Deep Learning Natural Language Processing*. 2021. arXiv: 2105.02388 [cs.CR]. URL: https://arxiv.org/abs/2105.02388.

[Idr+22] Badr Youbi Idrissi et al. *Simple data balancing achieves competitive worst-group-accuracy*. 2022. arXiv: 2110.14503 [cs.LG]. URL: https://arxiv.org/abs/2110.14503.

[Liu+22] L. Liu et al. "Investigating the Impact of Vulnerability Datasets on Deep Learning-Based Vulnerability Detectors". In: *PeerJ Computer Science* 8 (2022), e975. DOI: 10.7717/peerj-cs.975.

[MIT22] MITRE. *2022 CWE Top 25 Most Dangerous Software Weaknesses*. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. 2022.

[Sha+22] Deepak Kumar Sharma et al. "Explainable Artificial Intelligence for Cybersecurity". In: *Computers and Electrical Engineering* 103 (2022), p. 108356. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2022.108356. URL: https://www.sciencedirect.com/science/article/pii/S0045790622005730.

[Ver+22] Sahil Verma et al. *Counterfactual Explanations and Algorithmic Recourses for Machine Learning: A Review*. 2022. arXiv: 2010.10596 [cs.LG]. URL: https://arxiv.org/abs/2010.10596.

[Wu+22] Yueming Wu et al. "VulCNN: An Image-inspired Scalable Vulnerability Detection System". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 2365–2376. DOI: 10.1145/3510003.3510229.

[Che+23] Yizheng Chen et al. *DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection*. 2023. arXiv: 2304.00409 [cs.CR]. URL: https://arxiv.org/abs/2304.00409.

[CBK23] Roland Croft, M. Ali Babar, and Mehdi Kholoosi. *Data Quality for Software Vulnerability Datasets*. 2023. arXiv: 2301.05456 [cs.SE]. URL: https://arxiv.org/abs/2301.05456.

[Gan+23] Tom Ganz et al. "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*. Sophia-Antipolis, France: Springer-Verlag, 2023, pp. 282–301. ISBN: 978-3-031-37585-9. DOI: 10.1007/978-3-031-37586-6_17. URL: https://doi.org/10.1007/978-3-031-37586-6_17.

[GB23] Yuejun Guo and Seifeddine Bettaieb. "An Investigation of Quality Issues in Vulnerability Detection Datasets". In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, July 2023, pp. 29–33. DOI: 10.1109/eurospw59978.2023.00008. URL: http://dx.doi.org/10.1109/EuroSPW59978.2023.00008.

[HJH23] Haoyu He, Yuede Ji, and H. Howie Huang. *Illuminati: Towards Explaining Graph Neural Networks for Cybersecurity Analysis*. 2023. arXiv: 2303.14836 [cs.LG]. URL: https://arxiv.org/abs/2303.14836.

[Nij+23] Erik Nijkamp et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474 [cs.LG]. URL: https://arxiv.org/abs/2203.13474.

[Abb+24] Yassine Abbahaddou et al. *Bounding the Expected Robustness of Graph Neural Networks Subject to Node Feature Attacks*. 2024. arXiv: 2404.17947 [cs.LG]. URL: https://arxiv.org/abs/2404.17947.

[BW24] Samuel J. Bell and Skyler Wang. *The Multiple Dimensions of Spuriousness in Machine Learning*. 2024. arXiv: 2411.04696 [cs.LG]. URL: https://arxiv.org/abs/2411.04696.

[Chu+24] Zhaoyang Chu et al. "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. ACM, Sept. 2024, pp. 389–401. DOI: 10.1145/3650212.3652136. URL: http://dx.doi.org/10.1145/3650212.3652136.

[Dan+24] Seyed Shayan Daneshvar et al. *Exploring RAG-based Vulnerability Augmentation with LLMs*. 2024. arXiv: 2408.04125 [cs.SE]. URL: https://arxiv.org/abs/2408.04125.

[Din+24] Yangruibo Ding et al. *Vulnerability Detection with Code Language Models: How Far Are We?* 2024. arXiv: 2403.18624 [cs.SE]. URL: https://arxiv.org/abs/2403.18624.

[Liu+24] Shangqing Liu et al. *Enhancing Code Vulnerability Detection via Vulnerability-Preserving Data Augmentation*. 2024. arXiv: 2404.09599 [cs.CR]. URL: https://arxiv.org/abs/2404.09599.

[Mou+24] Esma Mouine et al. *Explaining the Contributing Factors for Vulnerability Detection in Machine Learning*. 2024. arXiv: 2406.03577 [cs.SE]. URL: https://arxiv.org/abs/2406.03577.

[Ste+24] David Steinmann et al. *Navigating Shortcuts, Spurious Correlations, and Confounders: From Origins via Detection to Mitigation*. 2024. arXiv: 2412.05152 [cs.LG]. URL: https://arxiv.org/abs/2412.05152.

[SAE24] Shaznin Sultana, Sadia Afreen, and Nasir U. Eisty. *Code Vulnerability Detection: A Comparative Analysis of Emerging Large Language Models*. 2024. arXiv: 2409.10490 [cs.SE]. URL: https://arxiv.org/abs/2409.10490.

[Ye+24] Wenqian Ye et al. *Spurious Correlations in Machine Learning: A Survey*. 2024. arXiv: 2402.12715 [cs.LG]. URL: https://arxiv.org/abs/2402.12715.

[ZZL24] Xin Zhou, Ting Zhang, and David Lo. *Large Language Model for Vulnerability Detection: Emerging Results and Future Directions*. 2024. arXiv: 2401.15468 [cs.SE]. URL: https://arxiv.org/abs/2401.15468.

[Ala+25] Fawaz Alanazi et al. "Enhancing cybersecurity vulnerability detection using different machine learning severity prediction models". In: *International Journal of Applied Science and Engineering* 22 (Jan. 2025), pp. 1–15. DOI: 10.6703/IJASE.202503_22(1).003.

[Gra25] Gradio Team. *Gradio – Build & share machine learning demos*. Accessed June 12,2025. 2025. URL: https://www.gradio.app/ (visited on 06/12/2025).

[Jha25] Amartya Jha. *The Ultimate Guide to Static Code Analysis in 2025 + 14 SCA Tools*. Accessed via CodeAnt AI "Static Code Analysis" blog :contentReference[oaicite:1]index=1. CodeAnt AI Blog. Feb. 2025. URL: https://www.codeant.ai/blogs/static-code-analysis-tools (visited on 06/09/2025).

[MIT25] MITRE. *CWE-20: Improper Input Validation*. https://cwe.mitre.org/data/definitions/20.html. Accessed: 2025-05-23. 2025.

[OWA25] OWASP Foundation. *Vulnerability Scanning Tools*. OWASP Community Page. 2025. URL: https://owasp.org/www-community/Vulnerability_Scanning_Tools (visited on 06/09/2025).

[Qia+25] Rui Qiao et al. *Group-robust Sample Reweighting for Subpopulation Shifts via Influence Functions*. 2025. arXiv: 2503.07315 [cs.LG]. URL: https://arxiv.org/abs/2503.07315.

[Whi25] Whirly Labs. *Joern – The Bug Hunter's Workbench*. 2025. URL: https://joern.io/ (visited on 06/05/2025).

[Wu+25] Tao Wu et al. *Understanding the Robustness of Graph Neural Networks against Adversarial Attacks*. 2025. arXiv: 2406.13920 [cs.LG]. URL: https://arxiv.org/abs/2406.13920.

Appendix A

Sustainable Development Goals

In 2015, United Nations member countries established a set of global objectives to promote human and technological development worldwide, resulting in the creation of a common agenda for 2030 that defines 17 Sustainable Development Goals. In alignment with this initiative, this project aims to contribute significantly to the following SDGs.

Figure A.1: Sustainable Development Goals (SDGs) aligned with this project: Industry, Innovation and Infrastructure (Goal 9), Peace, Justice and Strong Institutions (Goal 16), Quality Education (Goal 4), and Decent Work and Economic Growth (Goal 8).

- **Goal 4 - Quality Education**: The framework and methodology developed in this research offer valuable educational insights into the intersections of cybersecurity, explainable AI, and robust machine learning. The project facilitates deeper understanding of model behavior through visualization and interpretability tools, making it a useful resource for training the next generation of AI practitioners and security researchers.

- **Goal 8 - Decent Work and Economic Growth**: By improving the robustness and reliability of vulnerability detection tools, the project encourages responsible software development practices. The ability to identify true vulnerabilities, rather than spurious or misleading ones, leads to more secure software releases and efficient resource usage, reducing the cost and environmental impact associated with debugging, patching, and system failures.
- **Goal 9 – Industry, Innovation and Infrastructure**: The project promotes innovation in software security by advancing methods for automated vulnerability detection using machine learning, particularly Graph Neural Networks. By introducing counterfactual data augmentation, the work contributes to more reliable and interpretable AI systems. This supports the development of resilient digital infrastructure and fosters innovation in secure software engineering practices.
- **Goal 16 - Peace, Justice, and Strong Institutions**: Cybersecurity is a cornerstone of modern institutions, and this project enhances the trustworthiness of digital systems by mitigating spurious correlations in vulnerability detection models. Through robust and explainable AI tools, the work supports justice and transparency in automated code analysis, contributing to the protection of critical infrastructure and institutional integrity.

Appendix B

Resources

Aligned with the methodology and objectives outlined previously, the following hardware and software resources have been employed to implement, train, and evaluate the VISION framework. These tools provided a reliable and scalable technical foundation to conduct experiments in vulnerability detection and explainability using Graph Neural Networks.

Hardware

- **GPU**: Two NVIDIA RTX A4500 GPUs, each with 20GB memory, CUDA version 12.2, and driver version 535.183.01, were used to accelerate model training and evaluation, particularly for graph-based deep learning tasks.

Software

- **Python (v3.9)**: The primary programming language, chosen for its robust ecosystem in scientific computing and machine learning.
- **PyTorch and PyTorch Geometric**: Core frameworks for implementing and training GNN models such as Devign, with support for efficient graph operations and batch processing.
- **Illuminati Framework**: Used for post hoc GNN explanation and extended with a custom visualization module to support human-in-the-loop analysis.
- **Joern**: A static code analysis tool used to convert source code into Code Property Graphs (CPGs), enabling structural input for GNN processing.
- **OpenAI API**: Leveraged to generate high-quality counterfactual code examples, forming the basis of the augmentation strategy.
- **NetworkX and Matplotlib**: Visualization libraries employed to render graph structures and model attribution scores.
- **Visual Studio Code**: Used as the main development environment, offering integration with Git, linting tools, and debugging utilities.
- **GitHub**: Used for version control and collaboration, ensuring reproducibility and structured code management.

- **Virtual Environment Tools (e.g., venv)**: Used to isolate dependencies and ensure reproducible experiments.

Datasets

- **PrimeVul**: A publicly available, high-quality benchmark for software vulnerability detection.
- **CWE-20-CFA**: A custom-constructed dataset introduced in this work, derived from PrimeVul and augmented with validated counterfactuals.

Appendix C

Supplementary Code Listings

```python
def joern_parse(joern_cli_path, input_path, output_path, file_name):
    """
    Parses source code files into Joern's intermediate representation
    (Coded Property Graph - CPG).

    This function runs the `joern-parse` command-line tool to convert source code
    into a binary CPG file, which is later used for code analysis in Joern. The
    function executes the command using `subprocess.run` and returns the
    generated binary file's name.

    Parameters:
    - joern_cli_path (str): Path to the Joern CLI installation directory.
    - input_path (str): Path to the directory containing the source code.
    - output_path (str): Path where the parsed CPG binary file should be saved.
    - file_name (str): Base name for the output binary file (without extension).

    Returns:
    - str: The name of the generated binary file.
    """
    out_file = file_name + ".bin"
    # Subprocess calling
    joern_parse_call = subprocess.run(["./" + os.path.join(joern_cli_path, \
        "joern-parse"), input_path, "--out", os.path.join(output_path, \
            out_file)], tdout=subprocess.PIPE, text=True, check=True)

    return out_file
```

Figure C1: Code snippet for `joern_parse` function

```python
def joern_create(joern_path, in_path, out_path, cpg_file):
    """
    Executes a Joern script to extract function-level code property graphs (CPGs)
    and saves the results as a JSON file using direct communication with the
    process.

    Instead of writing the script to a file, this function sends commands directly
    to the Joern process via `stdin`. It:
    - Loads a previously generated CPG binary file.
    - Runs a predefined Joern script (`graph-for-funcs.sc`) to extract \
        function-level graph data.
    - Exports the results to a JSON file.

    Parameters:
    - joern_path (str): Path to the Joern CLI installation directory.
    - in_path (str): Path where the input CPG binary file is located.
    - out_path (str): Directory where the generated JSON file should be stored.
    - cpg_file (str): Name of the CPG binary file (e.g., "example.bin").

    Returns:
    - str: The name of the generated JSON file.
    """

    # Generate JSON output file name
    json_file = f"{cpg_file.split('.')[0]}.json"

    # Path for temporary script with commands
    if not os.path.exists("tmp"):
        os.mkdir("tmp")

    commands_script__path = os.path.abspath("tmp/joern_temp_script.sc")

    # Paths for script execution
    graph_script_path = os.path.abspath("joern/graph-for-funcs.sc")
    json_out = os.path.join(os.path.abspath(out_path), json_file)

    # Write commands to the script file
    with open(commands_script__path, 'w') as script_file:
        # Import CPG project
        script_file.write(f'importCpg("{os.path.abspath(in_path)}/{cpg_file}")\n')
        # Generate json graph
        script_file.write(f'cpg.runScript("{graph_script_path}").toString() |> \
            "{json_out}"\n')
        # Delete project
        script_file.write(f'delete("{cpg_file}")\n')

    # Set environment variables to avoid interactive mode
    env = os.environ.copy()
    env["JOERN_INTERACTIVE"] = "false"

    # Run Joern process and communicate via stdin (forcing non-interactive mode)
    joern_process = subprocess.Popen(
        [os.path.join(joern_path, "joern"), "--script", commands_script__path],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True,
        env=env,  # Pass modified environment variables
        bufsize=1,  # Enable line-buffering
    )

    # Send script commands to the process and close stdin
    outs, errs = joern_process.communicate(timeout=60)

    # Print any output or errors
    # if outs:
        # print(f"Outs: {outs}")
    # if errs:
        # print(f"Errs: {errs}")

    return json_file
```

Figure C2: Code snippet for `joern_create` function

```python
def process_dataset(df: pd.DataFrame, output_path: str):
    # Load existing dataset if it exists; otherwise, create an empty DataFrame
    if os.path.exists(output_path):
        dataset = pd.read_csv(output_path, index_col=0)
        print(f"Loaded existing dataset with {len(dataset)} examples.")
    else:
        dataset = pd.DataFrame(columns=['func', 'target', 'cwe', 'cpg'])

    with Progress() as progress:
        # Main Task (Total Work)
        main_task = progress.add_task(
            f"[magenta]Generating input examples dataset (0/{len(df)})...",
            total=len(df),
            bar_style="magenta"
        )
        # Create Sub-Task Once (Reused)
        secondary_task = progress.add_task(
            "[cyan]Example...",
            total=4,
            bar_style="cyan"
        )
        i = 0
        for index, example in df.iterrows():
        # for index, example in df.iterrows():
            # Check if this example has already been processed.
            matches = (dataset[dataset['func'] == example['func']]).index.tolist()
            if matches:
                idx_match = matches[0]
                progress.update(main_task, advance=1,
                    description=f"[magenta]Skipping already processed example ({i}/{len(df)})...")
                if index != idx_match:
                    dataset.loc[index] = dataset.loc[dataset.index == idx_match].iloc[0].copy()
                    dataset = dataset.drop(index=idx_match)
                i += 1
                continue

            retry_attempts = 0
            success = False

            while retry_attempts < MAX_RETRIES and not success:
                try:
                    progress.update(secondary_task, completed=0, description=f"[cyan]Generating example... \
                        (Attempt {retry_attempts+1})")

                    # Subtask 2: Code Parsing
                    progress.update(secondary_task, advance=1, description=f"[cyan]Parsing source code...")

                    # Save func as C file
                    source_file_path = os.path.join(PATHS["source"], f"{index}.c")
                    with open(source_file_path, 'w') as f:
                        f.write(example.func)

                    # Parsing function to .bin
                    cpg_file = joern_parse(JOERN_CLI_DIR, source_file_path, PATHS['cpg'], f"{index}_cpg")

                    # Subtask 3: Create CPG graphs JSON file
                    progress.update(secondary_task, advance=1, description=f"[cyan]Creating CPG with Joern...")
                    json_file = joern_create(JOERN_CLI_DIR, PATHS['cpg'], PATHS['cpg'], cpg_file)

                    # Subtask 4: Get CPG obj from JSON
                    progress.update(secondary_task, advance=1, description=f"[cyan]Processing CPG...")
                    graphs = json_process(PATHS['cpg'], json_file)
                    cpg = graphs[0][1]  # Get the CPG from graphs
                    example["cpg"] = cpg
                    example.to_pickle(os.path.join(PATHS['cpg'], f"{index}_cpg.pkl"))
                    progress.update(secondary_task, description=f"[green]CPG Completed.")

                    # Remove unused files
                    os.remove(os.path.join(PATHS['cpg'],f"{index}_cpg.bin"))
                    os.remove(os.path.join(PATHS['cpg'],f"{index}_cpg.json"))
                    os.remove(os.path.join(PATHS['cpg'],f"{index}_cpg.pkl"))
                    os.remove(os.path.join(PATHS['source'],f"{index}.c"))

                    # Add example
                    dataset = pd.concat([dataset, example.to_frame().T])
                    success = True

                except Exception as e:
                    retry_attempts += 1
                    progress.update(secondary_task, description=f"[red]Error occurred! \
                        Retrying ({retry_attempts}/{MAX_RETRIES})...")

                    if retry_attempts == MAX_RETRIES:
                        print(f"[ERROR] Failed to process example {index} after {MAX_RETRIES} attempts: {e}")

                        example["cpg"] = None
                        dataset = pd.concat([dataset, example.to_frame().T])

            # Update main task progress
            progress.update(main_task, advance=1, \
                description=f"[magenta]Generating dataset ({i+1}/{len(df)})...")
            i += 1
            # Mark as successful and exit retry loop
            success = True

            # Save dataset (every 10 data points)
            if not i % 100:
                dataset.to_csv(output_path)
                print(f"Saved dataset at {output_path}")
```

Figure C3: Code snippet for `process_dataset` function

```
def nodes_to_input(nodes, target, nodes_dim, keyed_vectors, edge_type):

    nodes_embedding = NodesEmbedding(nodes_dim, keyed_vectors)
    graphs_embedding = GraphsEmbedding(edge_type)
    label = torch.tensor([target]).float()

    x, code_embedding_mapping = nodes_embedding(nodes)
    edge_index=graphs_embedding(nodes)

    return Data(x=x, edge_index=edge_index, y=label), \
        code_embedding_mapping
```

Figure C4: Code snippet for `nodes_to_input` function

```
def process_cpg_to_nodes_row(row):
    cpg = eval(row.cpg)
    ordered_nodes, nodes_by_line_map = parse_to_nodes(cpg, NODES_DIM)
    return pd.Series({"nodes": ordered_nodes, \
        "nodes_by_line_map": nodes_by_line_map})
```

Figure C5: Code snippet for `process_cpg_to_nodes_row` function

```
def process_nodes_to_input_row(row, w2vmodel):
    input_series, code_embedding_mapping = nodes_to_input(row.nodes, \
        row.target, NODES_DIM, w2vmodel.wv, EDGE_TYPE)
    return pd.Series({"input": input_series, \
        "code_embedding_mapping": code_embedding_mapping})
```

Figure C6: Code snippet for `process_nodes_to_input_row` function

```python
for dataset in args.dataset:

        print(f"\nGenerating Input from CPG")
        print("----------------------------------------")

        dataset_path = "datasets/dataset/dataset_CWE-20.pkl"
        output_path = "datasets/dataset/dataset_CWE-20_input.pkl"

        if os.path.exists(output_path):
            dataset_df = pd.read_pickle(dataset_path)
            output_df = pd.read_pickle(output_path)
            df_init = True

            dataset_df = dataset_df[~dataset_df.index.isin(output_df.index)]

        else:
            df_init = False
            dataset_df = pd.read_pickle(dataset_path)

        total_examples = len(dataset_df)

        # Model initialization
        w2vmodel = Word2Vec(**WORD2VEC_ARGS)

        # Setup rich progress bar
        with Progress(
            TextColumn("[bold magenta]Processing {task.fields[dataset]} ({task.completed}/{task.total})..."),
            BarColumn(),
            TextColumn("[bold cyan]{task.percentage:>3.1f}%"),
            TimeRemainingColumn(),
        ) as progress:

            main_task = progress.add_task(f"[magenta]Processing {dataset.upper()} dataset",
                                          total=total_examples, dataset=dataset.upper(),)

            w2v_init = True
            i = 0
            for index, row_series in dataset_df.copy().iterrows():

                row_df = row_series.to_frame().T
                # Function Tokenization
                tokenized_func_df = tokenize(row_df)
                func_tokens = tokenized_func_df.tokens

                # Build and Train Word2Vec Model
                w2vmodel.build_vocab(corpus_iterable=func_tokens, update=not w2v_init)
                w2vmodel.train(func_tokens, total_examples=w2vmodel.corpus_count, epochs=1)

                # Embed cpg to node representation and pass to graph data structure
                row_df[["nodes", "nodes_by_line_map"]] = row_df.apply(process_cpg_to_nodes_row, axis=1)

                # remove rows with no nodes
                row_df = row_df.loc[row_df.nodes.map(len) > 0]

                # Apply the function and create both "input" and "map" columns
                row_df[["input", "code_embedding_mapping"]] = row_df.apply(lambda row: \
                    process_nodes_to_input_row(row, w2vmodel), axis=1)

                progress.update(main_task, advance=1)
                i += 1

                if not df_init:
                    output_df = row_df
                    df_init = True
                else:
                    output_df = pd.concat([output_df, row_df])

                if w2v_init:
                    w2v_init = False

                if i % EXAMPLES_PER_SAVE == 0:
                    output_df.to_pickle(output_path)
                    print(f"Saved dataset at {output_path}")
                    # Save Word2Vec model
                    w2vmodel.save('tmp/dataset/w2v/w2vmodel.wv')

            w2vmodel.save('tmp/dataset/w2v/w2vmodel.wv')
            output_df.to_pickle(output_path)
            print(f"Final dataset saved at {output_path}")
```

Figure C7: Code snippet for `process_cpg_to_input` function

```python
class IlluminatiExplainer(nn.Module):

    coeffs = {
        'edge_size': 0.005,
        'edge_reduction': 'sum',
        'node_feat_size': 1.0,
        'node_feat_reduction': 'mean',
        'edge_ent': 1.0,
        'node_feat_ent': 0.1,
        'mask_l1': 0.001,   # L1 regularization coefficient for sparsity
        'mask_var': 0.001,  # Variance regularization coefficient
    }

    def __init__(self, model, epochs: int = 50, lr: float = 0.01, agg1="max", \
        agg2="max", num_hops: int = None, device: str = "cpu", seed: int = SEED):

        super(IlluminatiExplainer, self).__init__()
        self.model = model
        self.epochs = epochs
        self.lr = lr
        self.__num_hops__ = num_hops
        self.device = device
        self.seed = seed
        # self.drop = nn.Dropout(p=0.2)
        self.drop = nn.Identity()
        self.model.to(device)

        # Additional learnable scaling parameters for the mask scores:
        # Temperature scaling for edge masks
        self.temp_edge = nn.Parameter(torch.tensor(1.0))
        # Temperature scaling for node feature masks
        self.temp_node = nn.Parameter(torch.tensor(1.0))
        # Power transform for edge masks
        self.power_edge = nn.Parameter(torch.tensor(1.0))
        # Power transform for node masks
        self.power_node = nn.Parameter(torch.tensor(1.0))

        # Define aggregation functions
        if agg1 == "mean":
            self.agg1 = torch.mean
        elif agg1 == "min":
            self.agg1 = torch.min
        elif agg1 == "max":
            self.agg1 = torch.max
        elif agg1 == "sum":
            self.agg1 = torch.sum
        else:
            self.agg1 = self.custom_agg  # use custom if no standard option fits

        if agg2 == "mean":
            self.agg2 = torch.mean
        elif agg2 == "min":
            self.agg2 = torch.min
        elif agg2 == "max":
            self.agg2 = torch.max
        elif agg2 == "sum":
            self.agg2 = torch.sum
        else:
            self.agg2 = self.custom_agg
```

Figure C8: Code snippet for `IlluminatiExplainer` class (Illuminati explainer implementation

```
def __set_masks__(self, data, node: bool = True, synchronize: bool = True, \
    edge_mask=None):
    # Set the seed for reproducibility
    self.__set_seed__(data)
    (N, F), E = data.x.size(), data.edge_index.size(1)
    num_nodes = N

    if edge_mask is not None:
        for module in self.model.modules():
            if isinstance(module, MessagePassing):
                module.__explain__ = True
                module.__edge_mask__ = edge_mask
        return

    std = 0.1
    node_feat_mask = torch.randn(1, F, generator=self.gen, device=self.device) \
        * std if not node else torch.randn(N, F, generator=self.gen, \
        device=self.device) * std
    edge_mask = torch.randn(E, generator=self.gen, device=self.device) * std
    std = torch.nn.init.calculate_gain('relu') * sqrt(2.0 / (2 * N))
    edge_mask = torch.randn(E, generator=self.gen, device=self.device) * std
    if not node and self.self_loop_mask is not None:
        edge_mask[self.self_loop_mask] = torch.ones(num_nodes)
    if node and synchronize:
        node_feat_mask = torch.mean(node_feat_mask, dim=-1, keepdim=True)
    self.node_feat_mask = nn.Parameter(node_feat_mask)
    self.edge_mask = edge_mask

    for module in self.model.modules():
        if isinstance(module, MessagePassing):
            module.__explain__ = True
            module.__edge_mask__ = self.edge_mask
```

Figure C9: Code snippet for `__set_masks__` `IlluminatiExplainer` function

```python
def explain_graph(self, data, loss_fc, node: bool = True, synchronize: bool = False):
    self.model.eval()
    self.__clear_masks__()
    (N, F), E = data.x.size(), data.edge_index.size(1)

    # Get prediction WITHOUT torch.no_grad() to allow gradients
    out = self.model(data.to(self.device))
    pred_label = out.detach()  # Retain real-valued prediction for explanation

    self.__get_indices__(data)
    self.__set_masks__(data, node=node, synchronize=synchronize)
    optimizer = torch.optim.Adam([self.node_feat_mask, self.edge_mask],
                                 lr=self.lr)

    for epoch in range(self.epochs):
        optimizer.zero_grad()
        node_feat_mask = self.__refine_mask__(self.node_feat_mask, beta=(epoch + 1) / self.epochs) if node \
            else self.node_feat_mask.sigmoid()
        h = data.x * node_feat_mask
        data_tmp = Data(x=h, edge_index=data.edge_index, batch=torch.zeros(N, dtype=torch.long))
        edge_mask = self.__refine_mask__(self.edge_mask, beta=(epoch + 1) / self.epochs) if node \
            else self.edge_mask.sigmoid()
        self.__set_masks__(data_tmp, edge_mask=edge_mask)
        out = self.model(data_tmp.to(self.device))
        loss = self.__loss__(out, pred_label, loss_fc)
        loss.backward()
        optimizer.step()

    node_feat_mask = self.__refine_mask__(self.node_feat_mask, training=False) if node \
        else self.node_feat_mask.sigmoid()
    edge_mask = self.__refine_mask__(self.edge_mask, training=False) if node \
        else self.edge_mask.sigmoid()
    node_mask = torch.zeros(node_feat_mask.shape[0])

    if node:
        node_feat_msg = torch.sum(node_feat_mask * data.x, dim=-1).view(-1)
        x = data.x.clone()
        x[x > 0.] = 1.
        node_feat_mask = node_feat_mask * x
        for n in range(N):
            idx = torch.nonzero(x[n])
            node_feat_msg[n] = self.custom_agg(node_feat_mask[n, idx])
        for n in range(N):
            if self.out_degree[n] > 0 or self.in_degree[n] > 0:
                out_masks = torch.zeros(1)
                if self.out_degree[n] > 0:
                    out_masks = edge_mask[self.out_edge_mask[n]]
                node_mask_out = out_masks * node_feat_msg[n]
                node_mask_out = self.agg1(node_mask_out)
                in_masks = edge_mask[self.self_loop_mask[n]] if self.self_loop_mask is not None \
                    else torch.zeros(1)
                node_mask_in = in_masks * node_feat_msg[n]
                if self.in_degree[n] > 0:
                    in_nodes = data.edge_index[0, self.in_edge_mask[n]]
                    in_masks = edge_mask[self.in_edge_mask[n]]
                    if self.self_loop_mask is not None:
                        in_masks = torch.cat((in_masks.view(-1), edge_mask[self.self_loop_mask[n]].view(-1)))
                    node_mask_in = in_masks * node_feat_msg[in_nodes]
                node_mask_in = self.agg1(node_mask_in)
                node_mask[n] = self.agg2(torch.cat((node_mask_in.view(-1), node_mask_out.view(-1))))
    else:
        node_mask = torch.zeros(N)
        for n in range(N):
            out_max = torch.tensor(0, dtype=torch.float)
            in_max = torch.tensor(0, dtype=torch.float)
            if self.out_degree[n] > 0:
                out_max = torch.max(edge_mask[self.out_edge_mask[n]])
            if self.in_degree[n] > 0:
                in_max = torch.max(edge_mask[self.in_edge_mask[n]])
            node_mask[n] = torch.max(out_max, in_max)

    node_mask = self.scores_scaling_transform(node_mask)
    self.__clear_masks__()
    return node_feat_mask, edge_mask, node_mask
```

Figure C10: Code snippet for `explain_graph` function

```python
def minimal_subgraph_by_adding(self, data, node_importance, model, threshold=0.5):
    """
    Greedily adds nodes (starting with the highest scoring) until the subgraph
    produces a prediction equal to the full graph's target prediction.
    In the returned submask (of shape [num_nodes]), the selected nodes are assigned
    their respective importance values (and zero elsewhere). In any case at least one node is returned.

    Parameters:
        data: The original graph Data object.
        node_importance: A tensor with importance scores for each node.
        model: The trained GNN model.
        threshold: Threshold for binary predictions.

    Returns:
        A tuple (submask, selected_nodes, confidence_score, ep) where:
            - submask is a tensor of shape [num_nodes] with the node importance values at
            the selected indices and zeros elsewhere.
            - selected_nodes is the list of indices in the candidate subgraph.
            - confidence_score is the final raw model output on the masked graph.
            - ep is 1 if the candidate's prediction matches the original target and 0 otherwise.
    """
    data = data.to(self.device)
    model.eval()
    with torch.no_grad():
        output = model(data)
        if output.dim() > 1:
            target_label = output.argmax(dim=1)[0].item()
        else:
            target_label = (output > threshold).long().item()

    sorted_indices = torch.argsort(node_importance, descending=True).tolist()
    selected_nodes = []
    confidence_score = None
    ep = 0  # essentialness: 1 if candidate prediction matches target, 0 otherwise.

    for idx in sorted_indices:
        if idx in selected_nodes:
            continue
        selected_nodes.append(idx)
        masked_data = self.mask_nodes_in_data(data, selected_nodes)
        with torch.no_grad():
            confidence_score = model(masked_data).cpu()
            if confidence_score.dim() > 1:
                pred = confidence_score.argmax(dim=1)[0].item()
            else:
                pred = (confidence_score > threshold).long().item()
        if pred == target_label:
            ep = 1
            break

    # Ensure at least one node is selected.
    if len(selected_nodes) == 0:
        selected_nodes = [sorted_indices[0]]
        masked_data = self.mask_nodes_in_data(data, selected_nodes)
        with torch.no_grad():
            confidence_score = model(masked_data).cpu()
        if confidence_score.dim() > 1:
            pred = confidence_score.argmax(dim=1)[0].item()
        else:
            pred = (confidence_score > threshold).long().item()
        ep = 1 if pred == target_label else 0

    submask = torch.zeros(data.x.size(0))
    submask[selected_nodes] = node_importance[selected_nodes]

    return submask, selected_nodes, confidence_score, ep
```

Figure C11: Code snippet for `minimal_subgraph_by_adding` function