Context-Driven Root Cause Analysis for Software Security Vulnerabilities

by

Md Iqbal Hossain Shuvo

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
October 2025

Abstract

This work introduces a context-aware causal inference framework designed to enhance both interpretability and robustness in automated vulnerability detection. The proposed method explicitly reconstructs root-to-sink causal chains that trace the propagation of vulnerabilities across functions, thereby revealing interprocedural spreading behaviors that conventional models overlook. Programs are encoded as heterogeneous graphs enriched with GraphCodeBERT representations, enabling a nuanced capture of syntactic and semantic dependencies. During inference, beam search with Adaptive Causal Contextualization (ACC) assembles executable causal chains that respect control- and data-flow reachability while maintaining cross-functional coherence and avoiding cycles. To further improve the interpretability of long, multi-function traces, a Causal Knowledge Graph (CKG) of frequently observed relation motifs is mined and employed as a weak structural prior, gently guiding beam expansion without compromising ACC's admissibility.

To rigorously evaluate the causal soundness of the proposed framework, two novel metrics are introduced. The Counterfactual Consistency Score (CCS) quantifies the stability of predictions under targeted causal perturbations, while the Causal Feature Attribution Measure (CFAM) assesses the alignment between the model's attention and the code elements that genuinely drive vulnerability outcomes. Together with traditional performance metrics, these measures establish a comprehensive evaluation protocol that validates both predictive effectiveness and causal grounding.

By transitioning from correlation-based analysis to mechanism-aware causal reasoning, this research advances the transparency, reliability, and practical utility of DL-based vulnerability analysis. The proposed framework not only enhances developers' ability to interpret model outputs but also provides a principled foundation for trustworthy, traceable, and targeted remediation in real-world software systems.

Acknowledgments

I would like to express my sincere gratitude to everyone who contributed to the successful completion of this thesis. First, I thank my Supervisor, Y M, for their unwavering support, invaluable guidance, and patience throughout this research journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Software now forms the foundation of critical infrastructure, commerce, and daily life. Cloud platforms manage sensitive data and services, while mobile, embedded, and cyber–physical systems extend software's reach into virtually every domain. As scope and connectivity expand, codebases grow in size and complexity, creating new opportunities for subtle security flaws. Undetected vulnerabilities threaten confidentiality, integrity, and availability, allowing unauthorized access, data leakage, and service disruption that can spread across dependent systems. Addressing these risks requires not only detecting suspicious patterns but also understanding the *mechanism* through which a flaw originates at a *root*, *propagates* through intermediate computations and function calls, and reaches an exploitable *sink*. Such mechanisms often cross function and module boundaries, where conventional analysis becomes difficult.

Traditional software assurance methods rely on two main lines of analysis: *static* and *dynamic*. Static analysis inspects source code without executing it, using rule-based reasoning, type systems, and data-flow frameworks to identify potential weaknesses. Dynamic analysis, by contrast, observes runtime behavior on real or simulated inputs to expose concrete failures. Each has strengths and limitations: static analysis covers many paths but can over-approximate, while dynamic analysis yields precise traces but with limited coverage. As codebases evolve in scale and heterogeneity, vulnerabilities that span argument→parameter bindings, return→caller flows, alias relationships, and branching conditions remain hard to track comprehensively using either technique alone.

To complement these classical approaches, learning-based methods have emerged to model source code as sequences, trees, or graphs. Graph-based learning methods have been particularly effective because they align naturally with program structure: *Abstract Syntax Trees* (ASTs), *Control-Flow Graphs* (CFGs), and *Data-Flow Graphs*

(DFGs) capture relational dependencies between statements, variables, and functions. When combined, these perspectives form the *Code Property Graph* (CPG), a unified representation of syntax, control, and data dependencies [55]. Building on such representations, neural models such as *Devign* aggregate multi-view code semantics and achieve strong benchmark performance in vulnerability detection [28]. Explanation approaches including *LEMNA* and *GNNExplainer* highlight influential features or subgraphs to interpret model predictions [16, 57]. Meanwhile, structure-aware pre-training approaches such as *GraphCodeBERT* enhance token embeddings with explicit data-flow information learned from large code corpora [15].

Despite these advances, current detectors often remain *statement-centric* or *subgraph-centric*. They can indicate *where* a vulnerability might exist but rarely reconstruct a complete, executable causal chain that explains *why* and *how* it emerges. Most models rely on statistical correlations observed in training data rather than stable causal mechanisms. As a result, they can be sensitive to benign code edits and show limited generalization to new projects or unseen code patterns [42, 56]. Recent work in causal and explainable vulnerability detection such as *COCA*, *Snopy*, *VulCausal*, *CausalVul*, and counterfactual explainers like *CFExplainer* has made progress in reducing spurious correlations and improving local interpretability [4, 21, 39, 9]. However, most existing models still stop short of reconstructing verified, interprocedural *root→propagation→sink* mechanisms that developers can trace and validate across real software systems.

1.2 Problem Statement

Despite significant advances in learning-based vulnerability detection, current detectors and localizers generally fail to reconstruct an explicit, executable, end-to-end causal pathway tracing from the initial vulnerability *root cause*, through intermediate *propagation* steps, to the final exploitable *sink*. While many methods leverage rich program graphs and structure-aware pretraining, their predictions primarily rely on statistical regularities and highlight local saliency, resulting in limited interpretability, weakened robustness to code changes, and reduced guidance for practical remediation. This issue becomes particularly acute for *interprocedural* vulnerabilities, whose enabling conditions span multiple functions, files, or modules, involving complex data and control flows such as argument-to-parameter bindings and aliasing relations [42, 41, 56].

Existing detectors flag risky code but rarely reconstruct how a flaw spreads across functions. This thesis addresses this limitation by explicitly building executable root→. . . →sink causal chains that capture inter-procedural vulnerability propagation. At inference time, candidate chains are explored using a constrained beam search guided by Adaptive Causal Contextualization (ACC), while a lightweight Causal Knowledge Graph (CKG) acts as a weak structural prior to promote coherent transitions without altering legality. Specifically, the framework focuses on recovering a single, plausible, and executable *interprocedural* causal chain for each detected vulnerability by unifying intra-procedural program structures (AST, CFG, DFG) with interprocedural semantics, including call-return relations, argument→parameter and return→caller bindings, and aliasing. The reconstructed chain must be consistent with feasible control and data paths, forming an executable sequence of events that faithfully explains how the vulnerability emerges and propagates across the software system.

1.2.1 Research Questions

1. **Causal Signals and Robustness.** Which *representational* choices—such as an augmented Code Property Graph (CPG) enriched with call-graph edges, argument-to-parameter mappings, return-to-caller links, and alias/points-to semantics and which *learning* mechanisms, including structure aware pretraining, graph attention networks, and causal regularization most effectively enhance *causal fidelity* while preserving robustness to benign code refactorings and generalizing across diverse software projects? The primary objective is to prioritize features that genuinely *enable* vulnerability exploitation, rather than those that exploit spurious statistical correlations.

2. **Inter-procedural Causal Chain Reconstruction.** How can context-aware graph reasoning methods *algorithmically* reconstruct *executable* causal chains that span from root cause through propagation paths to sink, while respecting interprocedural boundaries, adhering to control and data-flow feasibility constraints, and remaining minimal yet sufficient for comprehensive explanation? The aim is to produce developer-actionable causal chains that faithfully reflect actual program behavior across functions and modules.

3. **Rigorous Evaluation of Causal Explanations.** What principled criteria and evaluation protocols can rigorously quantify *causal faithfulness* and counterfactual robustness beyond conventional accuracy-based metrics, particularly for chain-centric explanations? Specifically, how should the *Counterfactual Consistency Score* (CCS) and the *Causal Feature Attribution Measure* (CFAM) be formally defined and instantiated, including intervention schemes, attribution alignment rules, and reporting standards to ensure evaluations are reliable, reproducible, and comparable across studies?

To answer these questions, this thesis proposes a chain-centric, interprocedural representation aligned with reasoning mechanisms that emphasize causally significant signals, supported by an evaluation framework that rewards explanations demonstrating counterfactual stability and attribution faithfulness. Together, these components enable faithful, interpretable, and robust reconstruction of vulnerabilities, especially those spanning functional and modular boundaries and establish a foundation for trustworthy and actionable automated vulnerability analysis.

1.3 Research Aims and Contributions

The overarching objective of this thesis is to design and validate a rigorous, *causally informed* framework for software vulnerability analysis that interprets vulnerabilities as *interprocedural mechanisms* rather than as isolated statement-level symptoms. Conventional program analysis techniques and contemporary machine learning approaches have made progress in detecting vulnerable code, yet both remain limited in their ability to reconstruct *how* and *why* vulnerabilities emerge and propagate through software systems. Traditional static and dynamic analyses face scalability and context-sensitivity issues, while deep learning methods frequently depend on statistical correlations that lack causal grounding. In practice, understanding a vulnerability requires uncovering the complete sequence of causal relationships from the *root cause*, through multiple transformation and propagation stages, to the ultimate *exploitable sink*. This thesis addresses that gap by establishing explicit causal connections that map how insecure data or logic moves through the program, why those flows occur, and what makes them exploitable. To achieve this goal, the proposed research develops (i) a representational foundation that supports executable reasoning across functions, files, and modules, (ii) a reasoning pipeline that emphasizes genuinely causal patterns over coincidental correlations, and (iii) evaluation criteria that measure the *faithfulness*, *stability*, and *actionability* of the system's outputs, not just their predictive accuracy.

1.3.1 Aims

This thesis advances three tightly connected research aims, each addressing a critical aspect of causally grounded vulnerability analysis.

Aim 1: Causal Fidelity and Robustness My aim focuses on ensuring that models learn *causally meaningful* signals rather than depend on superficial regularities in the data. Many deep learning vulnerability detectors learn patterns that correlate with certain types of insecure code but are not actually responsible for exploitability for instance, variable naming conventions or formatting artifacts. This thesis seeks to prioritize signals that truly *enable* vulnerabilities, such as untrusted input sources, data transformations, missing validation checks, and unsafe sink operations.

Emphasizing causal factors improves both interpretability and generalization. Models trained on genuine causal indicators become more resilient to benign software modifications, such as variable renaming, modular reorganization, or code refactoring. Consequently, causal fidelity enhances robustness to evolving codebases and promotes consistent performance across diverse projects and repositories [42, 56]. By designing models that explicitly encode causal dependencies between code elements, the research aims to produce detectors that can maintain stable predictions under ordinary development changes and effectively transfer knowledge across programming environments.

Aim 2: Mechanism Reconstruction The second aim is to reconstruct *complete, executable* causal chains that explain how vulnerabilities arise and propagate through a system. In brief my aim is to build mechanism-aware, inter-procedural root→sink chains using beam staging with adaptive causal contextualization, with a CKG prior to improve chain coherence and interpretability. Real exploits typically unfold as multi-step processes: a flaw at one point in the code (such as unvalidated user input) propagates through various computations or function calls before reaching an exploitable sink. Traditional models tend to provide local cues highlighting a few suspicious lines or a small subgraph, but they rarely piece together the full causal pathway.

This thesis addresses that limitation by introducing a method for systematic *mechanism reconstruction* that follows feasible control and data flows across the program. The approach traces relationships such as argument→parameter bindings, return→caller flows, and alias-induced dependencies to identify how data and influence travel interprocedurally. The reconstructed chains thus span functions, files, and modules, reflecting real-world attack surfaces where vulnerabilities often appear. These executable chains not only reveal where the most vulnerable code resides but also *why* it becomes vulnerable: they connect the initial trigger (root) to the intermediate transformations (propagation) and finally to the point where the vulnerability can be exploited (sink). This level of interpretability makes it much easier for developers and security analysts to validate, debug, and patch vulnerabilities [41, 36, 51].

Aim 3: Causal Evaluation The third aim focuses on developing new evaluation metrics that test the *causal faithfulness* of model predictions. Standard metrics such as accuracy, precision, recall, F1-score, and mean average precision are useful for classification but provide no insight into whether the model's reasoning correctly reflects real-world cause-and-effect relationships. This thesis introduces dedicated causal evaluation criteria that examine whether reconstructed vulnerability chains remain consistent and reliable under hypothetical changes or *counterfactual interventions*.

Specifically, the framework tests whether predictions remain stable when key causal elements are modified for instance, when an input sanitizer is added or strengthened, when a potentially dangerous function call is replaced with a safe alternative, or when an argument→parameter connection is removed. It also assesses whether the model's internal attribution its learned attention and feature weights aligns with ground-truth causal components of vulnerabilities. Together, these measures ensure that both detection and explanation are meaningful, actionable, and reliable in realistic, evolving software contexts [3, 9].

1.3.2 Contributions

Achieving these aims requires advances across multiple technical dimensions: representation, reasoning, evaluation, and empirical validation. This thesis therefore contributes a coherent framework designed to integrate these layers into a consistent causal analysis pipeline.

(C1) Chain-Centric Program Representation: At its foundation, the framework develops a unified graph representation built upon the *Code Property Graph (CPG)* abstraction [55]. The representation integrates three key structural perspectives: the *Abstract Syntax Tree (AST)* for syntactic structure, the *Control-Flow Graph (CFG)* for execution order, and the *Data-Flow Graph (DFG)* for value and dependency propagation. These intra-procedural components are augmented with interprocedural semantics, including call-graph connectivity, argument→parameter and return→caller bindings, call-site context propagation, and coarse alias and points-to approximations. This design enables the graph to represent how data and control flow through the entire codebase. Such a comprehensive view supports *executable mechanism tracing* allowing chains to be reconstructed from root to propagation to sink across function and module boundaries.

(C2) Causal Reasoning Pipeline for Chain Assembly: Built upon this rich representation, the thesis introduces a *causal reasoning pipeline* that connects structural representation with learning-based inference. The pipeline begins by embedding code elements using structure-aware models such as *GraphCodeBERT*, which encode both lexical semantics and data-flow relationships [15]. It then applies *Graph Attention Networks (GATs)* to propagate and contextualize these embeddings over heterogeneous graph structures. To further enhance interpretability, the pipeline incorporates an *Adaptive Causal Contextualization (ACC)* module that dynamically identifies, orders, and connects code features contributing causally to the model's vulnerability prediction. The ACC mechanism explicitly assembles each causal link—root, intermediate propagators, and sink—forming a single, coherent, and executable narrative of the vulnerability. In parallel, the framework integrates a *Causal Knowledge Graph (CKG)* mined from training graphs, capturing common relation motifs that serve as a weak structural prior during constrained beam decoding. This CKG-guided decoding improves the coherence and readability of inter-procedural causal chains while preserving ACC's admissibility constraints, including control/data-flow reachability, cross-function legality, and cycle avoidance. Additionally, per-hop rationales (model score and prior influence) are logged to enable step-wise, transparent explanations. Unlike traditional attention or saliency-based techniques that yield diffuse visual explanations, this unified approach produces structured, executable causal chains that developers can inspect and verify [28].

(C3) Causal Evaluation Metrics: To rigorously assess both accuracy and causal quality, the thesis defines two metrics: the *Counterfactual Consistency Score (CCS)* and the *Causal Feature Attribution Measure (CFAM)*. CCS evaluates the model's response to controlled interventions whether predicted vulnerability status changes appropriately when causal factors are modified. A model with high CCS demonstrates consistent reasoning and reacts predictably to causal changes in the source code. CFAM, in contrast, quantifies how closely model attention and learned feature importance align with ground-truth causal components, such as known input sources, propagation links, and sinks. When combined, CCS and CFAM provide a dual perspective on causal reliability: one behavioral (through intervention) and one interpretive (through attribution). Together, they extend the evaluation of vulnerability detection systems beyond surface-level correctness to examine the underlying logic of their explanations [3, 9].

(C4) Analysis: Finally, the thesis presents comprehensive empirical validation on the *ReposVul* dataset [48], analyzing repositories with explicit interprocedural structure. The proposed framework is rigorously compared against leading baselines, including *COCA* [3], under both conventional classification metrics and the newly proposed causal measures. The results show that the causally guided, chain-centric framework improves causal detection eficiency [4, 21, 39] and also produces more interpretable and stable explanations for developers across different projects.

These empirical outcomes support the central claim of the thesis: that interprocedural, causally informed models can deliver more trustworthy, generalizable, and explanatory vulnerability detectors than purely statistical alternatives. Together, the representational innovations, causal reasoning methodology, and evaluation framework form a coherent contribution toward the long-term goal of *explainable and dependable automated software security analysis*.

1.4 Methodological Overview

This thesis develops a chain-centric, heterogeneous multigraph that integrates *Abstract Syntax Trees* (AST), *Control-Flow Graphs* (CFG), and *Data-Flow Graphs* (DFG) within a *Code Property Graph* (CPG). The graph is enriched with interprocedural semantics, namely a call graph, argument→parameter and return→caller bindings, call-site context, and coarse alias or points-to approximations, in order to capture realistic data and control transfers across functions and modules [53]. Nodes and edges are initialized with structure-aware embeddings from *GraphCodeBERT*, which encode lexical regularities together with data-flow cues learned from large code corpora [15]. A graph attention encoder then aggregates neighborhood information across heterogeneous edges, and a *flow-consistency* regularizer encourages attention to concentrate along CFG-reachable def–use chains and matched call or return links, while de-emphasizing edges that violate feasibility or typing constraints.

On this encoded substrate, I encode programs as heterogeneous graphs enriched with pretrained code representations. At inference, I assemble causal vulnerability chains via beam search under *Adaptive Causal Contextualization* (ACC), which enforces structural legality and prevents spurious or infeasible hops. To further improve cross-functional coherence, I incorporate a lightweight *Causal Knowledge Graph* (CKG) prior, derived from relation motifs mined from training graphs. This prior gently guides decoding toward plausible inter-procedural transitions without altering ACC's admissibility checks or legality constraints such as control/data-flow reachability, cross-function validity, or cycle avoidance. The outcome is an executable root→. . . →sink chain that concretely explains how the vulnerability propagates across functions.

Building on this, the *ACC* procedure formalizes causal chain construction as a constrained path search over the augmented CPG with four requirements: (i) control and data-flow feasibility, including argument→parameter and return→caller bindings, (ii) proper call or return balancing, (iii) alias-consistent substitutions when necessary, penalized by uncertainty, and (iv) parsimony, that is, the exclusion of redundant hops that do not affect sink reachability. The result is an interprocedural root→propagation→sink chain that aligns with executable program behavior and is suitable for developer inspection and validation [28, 17].

For training, the model employs a supervised vulnerability detection objective at the function or repository level, complemented by auxiliary regularizers that promote attention concentration on call and return edges implicated in positive samples and encourage sparsity in selected subgraphs to yield concise mechanisms. During inference, the encoder produces attention distributions, ACC constructs the causal chain under beam search with CKG guidance, and the system outputs both the detection prediction and its mechanistic explanation. This regime naturally supports repository-level datasets where interprocedural structure is crucial for realistic evaluation, such as *ReposVul* [48], and enables fair comparisons with strong graph-based vulnerability analysis baselines, for example *COCA* [28, 3].

Evaluation encompasses conventional performance metrics and as well as two causality-oriented measures. The *Counterfactual Consistency Score* (CCS) quantifies prediction stability under principled interventions applied to key chain components, for example, strengthening or inserting sanitization on a propagation edge, severing an argument→parameter binding, or substituting a safer API sink. For a faithful causal mechanism, edits to causal elements should consistently attenuate or invert the model's vulnerability prediction, while edits to non-causal context should have marginal influence [3, 9]. The *Causal Feature Attribution Measure*

(CFAM) estimates the proportion of attribution mass—from attention or gradient-based importance scores—assigned to causal chain elements (sources, propagation links, and sinks) as opposed to off-chain context. When annotated ground-truth is available, CFAM measures direct alignment; otherwise, it evaluates internal consistency by contrasting ACC-selected elements with frequency-matched but infeasible alternatives [21, 39]. Together, CCS and CFAM assess the faithfulness and stability of reconstructed causal mechanisms beyond mere label agreement, complementing recent advances in robustness- and explanation-oriented vulnerability detection research [3, 4, 9, 6, 29].

1.5 Dataset and Evaluation Preview

The experimental evaluation leverages the ReposVul dataset, a repository-level benchmark specifically designed to minimize patch entanglement and emphasize explicit interprocedural structure. By organizing vulnerable and fixed revisions at the repository level with detailed call relationships and multi-granular dependency annotations, ReposVul provides realistic scenarios to trace vulnerability mechanisms that span multiple functions and modules. The experiments use three standard splitting, applying standardized preprocessing, stratified sampling, and controlled random seeds for comparability and reproducibility. Baselines include state-of-the-art graph neural network detectors and the causality-aware COCA framework [28, 3], each rigorously retrained on consistent splits. Evaluation combines traditional metrics with novel causal criteria: the Counterfactual Consistency Score (CCS) measuring prediction stability under purposeful chain modifications, and the Causal Feature Attribution Measure (CFAM) quantifying how well attribution focuses on genuine causal elements versus incidental context. Beyond standard detection metrics, I report chain centric diagnostics such as, average chain length, and inter-procedural span and include brief case studies that display hop level rationale for the constructed chains. Robustness tests verify invariance to benign code changes, while qualitative case studies on reconstructed causal chains validate explanation fidelity and remediation utility. This comprehensive protocol enables a rigorous comparison of chain-centric causal analysis against strong learning-based baselines, supporting not only detection performance assessment but also the evaluation of faithful, executable vulnerability explanations.

1.6 Thesis Outline

Note: The detailed outline will be finalized once all chapters are complete to ensure exact alignment with the final structure and contributions.

Chapter 2

Literature Review

The increasing reliance on machine learning (ML) to analyze source code has fundamentally transformed the software security landscape. The growing need for more reliable and interpretable vulnerability detection has driven extensive exploration of deep learning (DL) approaches. Architectures notably Graph Neural Networks (GNNs) operating over program graphs-promise scalable vulnerability detection and localization by learning structure-aware representations directly from code, often surpassing the performance of traditional static and dynamic analysis. However, despite this success, important challenges persist: difficulties generalizing across projects and distributions, high sensitivity to spurious correlations, limited support for interprocedural reasoning, and explanations that are insufficiently faithful for developer remediation. This chapter synthesizes the trajectory from early DL-based vulnerability detection and model explanation to current causality-aware approaches. I give particular attention to the development of pre-trained structural encoders and the emerging need for datasets that better reflect interprocedural mechanisms, ultimately crystallizing the unresolved gaps that motivate the core contributions of this thesis.

2.1 Theoretical Foundations and Background

Here I consolidates the key theoretical concepts and technical background necessary for understanding the research presented in this thesis. It distinguishes software bugs from security vulnerabilities, reviews classical static and dynamic analysis, details canonical code representations that enable structurally informed reasoning, explains interprocedural semantics for modeling flows across function boundaries, and motivates the incorporation of causality and counterfactual reasoning into security analytics. Where appropriate, references to recent surveys and representative systems are provided to anchor definitions and scope [6, 29].

2.1.1 Software Bugs and Vulnerabilities

A *bug* is a defect that causes a program to behave incorrectly or unexpectedly. Not all bugs threaten security. A *vulnerability* is a defect that an adversary can exploit to violate confidentiality, integrity, or availability. Distinguishing benign errors from exploitable vulnerabilities requires understanding not only the defect itself but also the *mechanism* by which it can lead to a successful attack.

This mechanism can be conceptualized as a chain of causation that begins at a *root* condition (e.g., untrusted input, unchecked buffer length, use-after-free) and *propagates* through assignments, parameter passing, return values, aliasing relations, and implicit flows induced by control dependence. The chain becomes exploitable when a tainted or unsafe state reaches a sensitive *sink*, such as a buffer write, command/SQL execution, deserialization routine, or privileged operation. Executability depends on dominance and post-dominance of guards, exceptional paths, resource lifetimes, and other feasibility constraints. This thesis therefore treats vulnerabilities as *root→propagation→sink* mechanisms to be reconstructed and validated end to end.

2.1.2 Classical Program Analysis: Static and Dynamic

Static analysis reasons about program behavior without execution. It uses dataflow frameworks and abstract interpretation to approximate reachable states across all paths. Precision is governed by *flow sensitivity* (does the analysis respect statement order?), *context sensitivity* (does it distinguish call sites or call histories?), *path sensitivity* (does it track path conditions across branches?), and *heap/field sensitivity* (does it distinguish fields and heap objects). Interprocedural static analysis requires constructing and resolving call graphs (e.g., class hierarchy analysis, rapid type analysis, points-to based resolution) and often employs summary-based propagation or IFDS/IDE-style solvers for scalable, distributive problems. Alias and points-to analyses are critical because they determine whether references may refer to the same memory, thereby controlling precision of def–use reasoning. Static analysis scales and can identify candidate roots, propagators, and sinks early, but coarse abstractions or under-modeled sanitization may cause false positives.

Dynamic analysis executes the program (or symbolic abstractions) to observe concrete behaviors. Fuzzing mutates inputs to trigger failures; sanitizers instrument code to detect memory errors and undefined behavior at runtime; concolic/symbolic execution uses path constraints to steer toward hard-to-reach states. Dynamic analysis yields concrete *witness traces* but faces input-space and path-explosion limits. In practice, static analysis can prioritize likely vulnerable chains, while targeted dynamic validation confirms executability. This complementarity is especially useful for interprocedural mechanisms that cross module boundaries and depend on calling contexts.

2.1.3 Code Representations for Security Analysis

Abstract Syntax Tree (AST). The AST encodes the hierarchical syntactic structure of source code, abstracting away punctuation to represent declarations, expressions, and statements. It supports parsing, symbol resolution, and scope management. By itself, the AST lacks explicit execution ordering and data provenance, limiting its utility for end-to-end vulnerability tracing.

Control-Flow Graph (CFG). A CFG represents the flow of control between basic blocks, including exceptional control. Analyses derive dominance/post-dominance, loop structure, and reachability—all necessary for judging whether guarding conditions actually protect sinks. Interprocedural CFGs (ICFGs) connect call sites to callees and returns to callers, modeling control transfer across procedures and enabling reachability checks across functions.

Data-Flow Graph (DFG). A DFG links definitions to uses (def–use chains), tracking value provenance across assignments, operations, parameter passing, and returns. Static Single Assignment (SSA) form simplifies reasoning by introducing ϕ-nodes at merges. Interprocedurally, argument→parameter and return→caller bindings extend provenance across calls, while alias/points-to facts determine when references share storage.

Program Dependence Graph (PDG). The PDG unifies data and control dependencies in a single graph, enabling slicing along both axes and reasoning about implicit flows. PDGs are commonly intra-procedural; explicit extensions are required to model interprocedural flows for whole-program reasoning.

Code Property Graph (CPG). The CPG combines AST, CFG, and DFG into a heterogeneous, typed multigraph, supporting joint queries over syntax, execution order, and data dependencies [55]. CPGs are well suited to vulnerability discovery because they admit taint-style queries that trace feasible paths from sources through transformations to sinks while retaining syntactic anchors for precise localization. For interprocedural analysis, the CPG is enriched with call-graph edges, argument→parameter and return→caller links, call-site context (e.g., dynamic dispatch), and coarse alias information, enabling faithful reconstruction of cross-boundary propagation. This thesis adopts an augmented CPG as the backbone for chain-centric mechanism tracing.

Interprocedural Semantics. Executable vulnerability chains often cross function boundaries. To capture these flows, the representation integrates (i) a call graph, (ii) argument-to-parameter and return-to-caller relations, (iii) call-site contexts (e.g., receiver/dispatch in object-oriented code), and (iv) alias/points-to approximations. Together they support tracing explicit data flows and implicit control effects across modules in a way that aligns with feasible program execution.

2.1.4 Learning over Structure and Causality-Aware Analysis

Learning-based methods complement classical analysis by operating directly over structured code representations. Graph-based detectors leverage multi-view structure to classify or localize vulnerabilities; for example, statement- and line-level GNNs incorporate structural dependencies to improve localization [17, 6]. Structure-aware pretraining, such as GraphCodeBERT, injects data-flow relations into transformers and has been shown to improve cross-project generalization by learning context-sensitive embeddings from large code corpora [15, 12].

Despite these advances, empirical studies report degradation under benign refactorings and cross-repository evaluation, indicating sensitivity to spurious correlations and project-specific artifacts [42, 56]. This motivates *causality-aware* analysis that emphasizes genuine mechanisms over surface correlations. Recent approaches instantiate this principle in different ways. *VulCausal* and *CausalVul* introduce structural causal models and do-calculus to adjust for confounders and prioritize causal features [21, 39]. *Snopy* bridges change-based denoising (guided by vulnerability-fixing commits) with causality-aware graph attention to suppress vulnerability-irrelevant features [4]. *CFExplainer* adopts counterfactual reasoning for GNN-based vulnerability detection, identifying minimal graph edits that flip the model prediction and thereby elucidate decision-critical structures [9]. *COCA* combines combinatorial contrastive learning with dual-view (factual/counterfactual) inference to produce concise, decisive statements and to improve robustness [3]. Work on explainability and contributing-factor assessment further highlights the gap between correlation-based rationales and truly causal explanations, underscoring the need for evaluation that measures faithfulness and intervention stability [32, 24, 25].

Taken together, this background motivates a chain-centric, causally informed approach: adopt a unified, interprocedural representation (augmented CPG), reason over it with structure-aware models to prefer causally meaningful signals, and evaluate with criteria that test counterfactual stability and attribution alignment. The remaining chapters operationalize these ideas in a framework that reconstructs explicit *root→propagation→sink* mechanisms and assesses their fidelity alongside conventional performance metrics.

2.1.5 Causal Knowledge Graphs (CKG)

The concept of a *Causal Knowledge Graph (CKG)* has emerged in recent literature as a means of capturing the underlying mechanisms that explain *why* specific structural or semantic relationships tend to follow one another, rather than merely documenting that they co-occur. Within program analysis, such knowledge graphs allow models to represent and reason about directional causal dependencies among program relations. In contrast to conventional graph structures that store entities and their factual relations, a CKG encodes tendencies or patterns that indicate causal influence, thereby providing a foundation for more interpretable and mechanism-aware reasoning.

In the context of vulnerability detection, this thesis adopts the notion of a CKG to represent causal pathways that underlie inter-procedural vulnerability propagation. The nodes in the CKG correspond to relation types present in the heterogeneous program graph, such as function calls, argument-to-parameter bindings, return-to-caller edges, or data-flow and control-flow links. Directed links between these relation types indicate empirically observed transitions where one relation type tends to give rise to another during the construction of vulnerability chains. Beyond simple one-step relationships, the CKG also summarizes multi-step structures or *motifs* that frequently characterize real-world vulnerability spread. For example, a common motif in source code may involve a chain of relations where a function call is followed by argument propagation, and then by a return linkage, representing a full cross-function vulnerability transfer.

The value of such a causal structure lies in its ability to guide learning systems away from brittle, correlation-driven associations. Purely data-driven detectors often rely on local regularities in training data and may therefore overfit to patterns that are statistically strong but mechanistically invalid under domain shift. A CKG, estimated post hoc from the distribution of relations observed in training graphs, provides a lightweight structural prior that gently nudges the model toward transition patterns consistent with realistic causal propagation. By doing so, it helps reduce uncertainty during inference, encouraging chain assembly that is coherent and interpretable without depending on rigid rules or handcrafted heuristics.

To construct the CKG, relational evidence from training programs and derives a set of statistically frequent transition patterns between relation types. These patterns can be extended to bi- or tri-gram motifs that offer a compact summary of how causal relations typically unfold across inter-procedural contexts. The estimated graph thus not only represents likely cause–effect transitions but also preserves interpretable motifs that can later be inspected or visualized by developers to understand model behavior. Importantly, the CKG remains lightweight and non-intrusive: it is designed as an auxiliary source of structural plausibility rather than a hard constraint. During model decoding, evidence learned by the underlying detection model remains the primary decision factor, while the CKG acts as a weak regularizer that rewards transitions aligned with historically observed causal structures.

The integration of a CKG into vulnerability reasoning aligns with a growing body of research in explainable artificial intelligence and causal inference, which emphasizes interpretable mechanisms over purely correlative statistical accuracy. Prior work in software vulnerability analysis has highlighted the limitations of models that lack causal grounding, noting that explanations derived from attention or gradient-based saliency can easily misrepresent the true reasons behind a prediction. By explicitly encoding directional causal patterns among structural relations, the CKG contributes interpretability and stability to the chain reconstruction process. Each predicted causal link can be contextualized within a recognized relational motif, allowing developers to inspect and validate specific propagation paths.

Overall, while traditional knowledge graphs serve primarily as repositories of factual relationships, a CKG distinguishes itself by focusing on mechanistic tendencies that support reasoning under hypothetical interventions addressing questions such as what would happen if a specific relation were promoted or removed. This property makes CKGs particularly suitable for building and evaluating executable, root-to-sink vulnerability chains, where understanding the underlying propagation mechanism is as crucial as predicting the existence of a flaw itself.

2.1.6 Constrained Decoding and Beam Search for Structured Reasoning

Beam search has long been recognized as a widely used heuristic for exploring structured search spaces where the goal is to efficiently identify the most promising candidate solutions. It maintains a finite set of the top-ranked partial hypotheses at each expansion step, thereby balancing exploration and computational tractability. Within structured reasoning and program analysis, however, traditional unconstrained beam search, or naïve search strategies such as best-first or depth-first traversal, frequently encounter difficulties when dealing with validity constraints inherent to logical or semantic domains. These basic strategies can drift into infeasible or inconsistent paths that may appear locally optimal but ultimately violate global program semantics.

To address this limitation, the notion of *constrained decoding* also referred to as guided beam search has emerged as a powerful strategy for structured prediction tasks. Unlike standard decoding, constrained beam search dynamically filters or adjusts candidate expansions based on a combination of hard structural constraints and soft structural preferences. In the context of program graph reasoning, constraints may include control- and data-flow reachability conditions, interprocedural legality rules that govern how function calls and returns are matched, cycle avoidance, recursion depth control, and alias consistency requirements. These constraints ensure that the generated paths reflect feasible execution behavior and retain logical coherence across procedural boundaries.

Existing studies have applied constrained or guided beam search across various reasoning tasks such as syntactic parsing, symbolic execution, and knowledge graph inference. Many of these approaches rely on rule-based or manually designed heuristics to enforce validity, while some leverage scoring functions based on learned local evidence. However, pure heuristic control often fails to generalize well, and purely learned scoring can succumb to spurious correlations. Recognizing this gap, more recent work seeks to integrate statistical structure priors derived from domain relevant data into the decoding process, enabling the search to blend empirical regularities with deterministic legality checks.

Compared with more global optimization techniques such as Integer Linear Programming (ILP) or Satisfiability Modulo Theory (SMT) solving, constrained beam search offers a favorable trade off between interpretability, tractability, and output diversity. It allows efficient exploration of multiple plausible causal chains, providing a ranked set of candidate explanations that developers can empirically verify or analyze. The inclusion of legality checks and causal priors yields a decoding process that is both computationally efficient and mechanism-aware, aligning with the broader objective of producing structured, executable explanations that accurately capture vulnerability propagation in software systems.

2.2 Related Works

2.2.1 Static Analysis: Foundations and DL Integration

Static analysis examines software artifacts without executing them, reasoning exhaustively about possible program behaviors through frameworks like lattice-based dataflow analysis and abstract interpretation [55, 6]. Its precision depends on sensitivities to control flow, call context, execution paths, and heap or field representations. Interprocedural static analysis demands accurate call graph construction, supported by methods such as Class Hierarchy Analysis, Rapid Type Analysis, and points-to analyses to resolve aliasing [1, 29].

These analyses utilize canonical program representations: Abstract Syntax Trees (ASTs) encode syntactic structure; Control-Flow Graphs (CFGs) model executable order and branching; Data-Flow Graphs (DFGs) trace variable value definitions and uses; and Program Dependence Graphs (PDGs) unify data and control dependencies for program slicing. The Code Property Graph (CPG) integrates AST, CFG, and DFG into a heterogeneous multigraph that supports global queries essential for interprocedural taint tracking from sources to sinks [55].

Despite early identification benefits, classical static analysis suffers from false positives due to over-approximation in modeling sanitizers, aliasing, and dynamic dispatch [11, **?**]. Automated static analyzers such as Flawfinder, RATS, CPPCheck, SpotBugs, and PMD show variability in detection accuracy and false positive rates across languages like C++ and Java [**?**]. These limitations motivate the development of DL-augmented static analysis tools. For example, IRIS incorporates large language models (LLMs) with static analysis to infer taint specifications and improve vulnerability detection coverage and false discovery rates, surpassing traditional tools like CodeQL [27].

Further, hybrid approaches blend static analysis with ML classifiers to filter and prioritize warnings, addressing false positives and improving scalability [**?**]. Static analysis remains foundational, providing structured input to DL models, yet its limitations in handling complex, evolving, and domain-specific codebases remain a persistent challenge [1, 6].

2.2.2 Dynamic Analysis: Execution Evidence and DL Integration

Dynamic analysis observes real or symbolic execution of programs with concrete inputs to detect vulnerabilities manifesting during runtime [54, 53]. Techniques include fuzzing, which automates input mutation to explore program paths; sanitizers, which instrument binaries or source to catch memory errors; and symbolic or concolic execution, which employs solvers to systematically explore feasible paths leading to fault states.

Dynamic methods provide high precision witness traces, essential for actionable debugging, but suffer from path explosion and limited environmental coverage, particularly in large interprocedural contexts [54]. This gap has incentivized hybrid strategies integrating machine learning to guide input generation or prioritize suspicious execution traces, enhancing coverage efficiency [**?**].

Recent work employs reinforcement learning and deep models to optimize fuzzing or dynamic analysis workflows, combining symbolic constraints and learned heuristics to uncover hard-to-detect vulnerabilities [**?**]. These advances promise better balancing precision and exploration in complex, modern software systems.

Dynamic analysis's concreteness complements static analysis's scalability, and their integration with machine learning techniques is an active research frontier poised to improve vulnerability detection coverage, efficiency, and interpretability.

2.2.3 Deep Learning Based Techniques

Deep learning is transforming many areas of computer science and has been applied in research projects. In particular, graph neural networks (GNNs) [17], deep learning models have rapidly shown their promise in vulnerability detection, achieving amazing precision in identifying susceptible code patterns [5, 29, 60]. GNNs offer a natural approach for capturing complex software dependencies by modeling code as graphs, using nodes for elements (e.g., variables, functions) and edges for relationships (e.g., data flow, control flow). Hin et al. [17], for instance, showed how well GNNs identified vulnerabilities at the statement level. With the capacity to learn from the structure and semantic information of code, the models were able to spot vulnerabilities suggestive of buffer overflows or injection issues. Although many of these early deep learning models functioned as "black boxes," impairing the knowledge of their decision-making processes [26, 34], even with their precision. Especially in the context of software security, the ability to understand the reason *why* a model finds a piece of code as vulnerable is crucial for developers to implement fixes [46]. This lack of transparency is a major concern. Furthermore, impeding the acceptance of these approaches in practical software development processes is their difficulty in explaining model predictions. The US Government's executive order [46] on the safe, secure, and trustworthy development and use of artificial intelligence emphasizes how much artificial intelligence (AI) is becoming relied upon in important systems, where erroneous or untrustworthy predictions can have severe consequences.

Zhou et al. [28] presented a comprehensive comparative analysis of factors influencing the performance of deep learning (DL)-based vulnerability detection systems. Recognizing the complexity of software vulnerabilities, the authors sought to systematically evaluate how different design choices affect detection efficacy. To this end, they constructed two distinct datasets, capturing both data reliance and control dependencies extracted from programs, encompassing a diverse set of 126 different vulnerability types. Their methodology involved assessing the quantitative impact of several key elements on vulnerability identification performance. This included employing techniques for handling unbalanced data, incorporating control dependence information within code representations, and experimenting with various neural network architectures. To facilitate their analysis, Zhou et al. built a DL-based vulnerability detection system leveraging Joern, an expanded open-source code parser. The primary contribution of this work lies in its systematic assessment of the quantitative influence of these elements on vulnerability detection efficacy. The study's results offer valuable insights into the design and development of effective deep learning-based vulnerability detection systems. However, it's important to note that this research did not specifically address causal deep learning (CDL) or explicitly incorporate causal reasoning principles. The authors themselves highlighted the need to accommodate a wider range of vulnerability types, enhance the representation of code features, and overcome the limitations of relying solely on control and data dependencies.

Li et al. [42] performed empirical research on deep learning (DL) models for vulnerability detection. On Devign and MSR data, they polled and replicated nine state-of-the-art (SOTA) models. They looked at and examined model capabilities (agreement, variability, performance on several vulnerability categories, and difficulties in addressing particular code aspects). They investigated how model performance responded to project mix and training data size. They identified significant code aspects applied for prediction using model explanation tools. The main contribution of the writers was a thorough investigation of models of deep learning vulnerability detection. For assessing and contrasting several deep learning models for vulnerability identification, this research offers a useful standard. Still, this study paid little attention to causative factors. The writers urged more investigation on code patterns, possible inclusion of causal detection for better generalization, and addressing of the shortcomings of present model interpretation techniques.

Zou et al. [30] tackled this problem with domain adaptation methods and deep learning. To reduce distribution discrepancies between domains, their CD-VulD system learnt cross-domain representations. Learning token embeddings for generalization across tokens, the CD-VulD system transforms software program representations into token sequences. Abstract high-level representations based on those sequences are built using a deep feature model. Minimizing the distribution divergence between the source and destination domains helps to train cross-domain representations using the metric transfer learning framework (MTLF). The key contribution of the authors was a CD-VulD system learning cross-domain representations of source code via domain adaptation and deep learning. Practical vulnerability identification depends on generalizing across several fields, as models trained on one dataset might not work on another. This research did not, however, use causal reasoning; the authors observed constraints in evaluation scope and dependency on particular architectures.

2.2.4 Causal Reasoning and Causal Deep Learning Techniques

Conventional deep learning techniques have shortcomings, including their capacity to detect erroneous correlations and changes in dataset distribution. Stronger and broader methods have emerged from this. VulCausal, developed by Kuang et al. [21], solves what caused what in neural network models. Finding and eliminating bogus links between API functions, user-defined names, and code structure is VulCausal's primary objective. The authors provide a structural cause model to demonstrate how discovering vulnerabilities is affected by user-defined names, API library IDs, and code structure. The study employs covert adjustment in the reasoning stage to eliminate erroneous correlations and minimize the impact of confounders. By modeling and lowering spurious correlations, VulCausal aims to make vulnerability identification more accurate and consistent than conventional deep learning approaches. This is a crucial first step toward exposing vulnerabilities. The writers mostly included a causal perspective to eliminate misleading correlations and provide more accurate and reliable vulnerability detection. When causal inference techniques were developed, they connected with a whole new level and helped us to better understand the factors causing vulnerability in individuals. One significant fresh concept is the application of backdoor correction and causal reasoning approaches. The approach wasn't flawless, the writers noted, though, as it lacked scalability to extremely big codebases, it wasn't ubiquitous across many computer languages, and it couldn't show direct causality (beyond performance benefits). This indicates that in these fields additional research is required. One should also consider the extent of labour causal inference techniques demand and the requirement of well crafted models. Zelikman et al. [59] on self-taught optimizers also indicate that individuals are still striving to create machine learning models that are more efficient and helpful.

Le et al. [41] investigated how well existing deep learning-based detectors could handle vulnerabilities spanning several base units of code (MBUs). Their empirical evaluation of current deep learning-based detectors revealed a clear deficiency in tackling MBU vulnerabilities, usually overstretching accuracy by focusing on individual base units (IBUs). Although it did not specifically include causal reasoning in the detection technique, this research underlined the need of investigating the features and distribution of MBU vulnerabilities as well as the limits of current datasets. Together with a detailed analysis of their incidence and detection accuracy in modern deep learning (DL)-based detectors, the authors gave a description and categorization of MBU vulnerabilities. The key contribution of the authors was the proposal of a framework for the correct integration of MBU vulnerabilities in DL-based detection. The study emphasizes the significance of evaluating the spectrum of vulnerabilities and of studying detection methods. The problem of MBU vulnerabilities is particularly relevant in modern software development, as code usually involves numerous files and modules. Furthermore, improving the understanding of vulnerability types and their frequency is the research by Nong et al. [36] on authentic vulnerability generation through pattern mining and deep learning and the study by Woo et al. [51] on identifying 1-day vulnerabilities in reused open-source software components.

Cao et al. [4] proposed Snopy, a deep learning (DL)-based technique bridging sample denoising with causal graph learning to enhance vulnerability identification. Snopy expressly included causal reasoning utilizing a Causality-Aware Graph Attention Network (CA-GAT), a Feature Caching Scheme (FCS), and a Causality-Aware Graph Attention Network (CA-GAT) identifying bogus features. Though areas for future research remain, including generalizability to other programming languages and vulnerability types [51] and more sophisticated ways for mitigating spurious correlations in code, this approach represented a major step toward causal vulnerability detection. Initially, deleting vulnerability-irrelevant code elements and constructs using change-based sample denoising and then developing a Causality-Aware Graph Attention Network (CA-GAT) utilizing Feature Caching Scheme (FCS), Snopy learns causal vulnerability traits. The main contribution of the authors was a change-based method guided by vulnerability-fixing commits (VFCs) automatically removing vulnerability-irrelevant code components. A unique addition is made by the combination of sample denoising with causal graph learning; the usage of VFCs offers a moral approach to finding and eliminating pointless code components. Building strong and dependable vulnerability detection systems depends on one being able to differentiate between causal and spurious aspects.

Ganz et al. [14] focused on addressing data quality, model interpretability, robustness, and contextual sensitivity, thereby enhancing the usefulness of machine learning (ML)-driven vulnerability identification. This research particularly applied causal learning approaches to reduce confusing effects and improve detection robustness. They developed datasets by using a novel neural code augmentation technique. They assessed explanation tactics using a novel approach based on dynamic program analysis. They assessed models on their acquired biases using a novel assessment system based on causal learning. The main contribution of the authors was to provide strategies raising the relevance of learning-based vulnerability detection in practical environments. The study underlines the need to attend to pragmatic issues such as data quality and model interpretability. Relevant to the problems of model interpretability and explainability is the research by Suneja et al. [43] on evaluating model signal-awareness by means of prediction-preserving input minimization, together with the study by Yu et al. [58].

Growing interest in causal deep learning (CDL) results from the limits of current approaches, especially in terms of generalization and the capacity to find the underlying causes of vulnerabilities. By means of CausalVul, Rahman et al. [39] directly addressed the lack of resilience and generalization to out-of-distribution (OD) data in deep learning (DL)-based vulnerability detection. Explicitly leveraging do-calculus and the backdoor criterion, this two-stage approach aimed at identifying and eliminating false features using causal learning algorithms. This paper clearly shows a clear improvement in causal deep learning (CDL) for vulnerability detection, therefore highlighting the possibilities of causal inference to increase model generalization. The correlations between code features and the vulnerability label were shown by the authors using a causal graph. They then trained models that were less dependent on spurious features and more focused on causal features by the use of do-calculus and the backdoor criterion. The main contribution of the authors was to explicitly address false correlations, thereby introducing a fresh method to increase the generalization and resilience of deep learning (DL)-based vulnerability detection. Two important developments are the application of the backdoor criterion and do-calculus. Real-world applications depend on generalizing OOD data since the spread of vulnerabilities may evolve with time.

Chu et al. [9] addressed the lack of explainability in Graph Neural Networks (GNNs) for vulnerability identification. Using counterfactual reasoning, a type of causal inference, CFExplainer aimed to identify minimal changes to the input code graph that would influence the GNN prediction. This improved explainability by focusing on behaviours that alter the outcome and so provides a "what-if" analytical capability. The approach finds a minimum disruption in the code graph by inverting the prediction of the GNN. This provides developers with useful knowledge and guides them on the necessary changes to eliminate a vulnerability. The authors mostly contributed by developing a counterfactual explanation method for GNN-based vulnerability discovery. Counterfactual thinking helps developers find a more reasonable and workable argument. A better basis for understanding the application of counterfactual reasoning in explainable artificial intelligence is provided by the work of Lucic et al. [31].

Islam et al. [18] presented T5-GCN for vulnerability categorization, localization, and root cause identification. Although the "root cause" was sought for, this research did not specifically use causal deep learning (CDL), causal inference, or causal reasoning methods. Conversely, the explainability part of T5-GCN aims to identify the "root cause" of vulnerabilities, therefore indirectly guiding knowledge of the elements generating the vulnerability. The approach uses DeepLift-SHAP attribution values to determine the relevance of tokens in the code, therefore identifying the basic cause. Along with their classification, location, and a brief static description, the writers mostly contributed a method employing explainable methodologies to identify the root cause of a vulnerability. One original method is to combine GCNs and LLMs. Large language models (LLMs) for code analysis are a fast-expanding field of research; the research of Zelikman et al. Furthermore underlined in [59] on self-taught optimizers, the potential of LLMs in this field is relevant for the application of LLMs in code analysis, and vulnerability detection is the effort of Pearce et al. [37] to assess the code contributions' security of GitHub Copilot.

Cao et al. [3] presented Coca, a framework to enhance the causality and robustness of Graph Neural Networks (GNNs)-based vulnerability detection systems. Coca used dual-view causal inference, that is, factual and counterfactual reasoning, to pinpoint code statements most likely to be decisive for vulnerability discovery. This method showed a sophisticated use of causal ideas to solve constraints in robustness and explainability observed in past GNN-based detectors. It also underlined the difficulties in juggling concision with effectiveness in explanations. Coca trains GNNs less prone to false correlations and more focused on real vulnerability traits by means of combinatorial contrastive learning. The Explainer component generates succinct and powerful explanations using dual-view causal inference. Using supervised constrastive learning, the system is taught to identify the bug in all versions and distinguish between buggy and non-buggy code. It discovers a flaw and then employs factual and counterfactual thinking. This is a framework enhancing the causality and dependability of GNN-based vulnerability detection systems.

2.2.5 Explainability and Interpretability Techniques

In the first attempt to tackle the explainability issue, scientists aimed to create techniques that would reveal the inner workings of deep learning models, hence enabling more reasonable and reliable predictions. Proposed by Le et al. [26], GAVulExplainer was one of the first attempts to handle this using a model-agnostic approach, that is, one can apply GAVulExplainer to several deep learning models and genetic algorithms to help find important subgraphs causing vulnerabilities. The basic concept is to build a subgraph emphasizing the important elements causing the vulnerability, therefore offering a more understandable justification for the predictions of the model. While avoiding local optima, the genetic algorithms effectively seek accurate substructure information. This method employs a fidelity metric to assess the quality of produced explanations and lets users regulate the size of the explanation subgraph. GAVulExplainer marks a significant progress in enhancing the interpretability of GNN-based vulnerability detection, but it still lacks explicit modeling of the fundamental *causal* links between code characteristics and vulnerabilities. Finding contributing subgraphs still takes front stage without exploring causal deep learning (CDL) methods. Furthermore, its assessment depends on fidelity criteria, which might not completely reflect the pragmatic value of the explanations for developers in real-world debugging situations, especially given the complexity of software systems and the several ways vulnerabilities might show themselves. This study addressed the demand for explainable prediction since they realized that successful remedial action and developer confidence depend on knowing the "why" behind the prediction of a model.

Li et al. [26] presented that Vulanalyzer is another significant initiative aimed at improving explainability. Specifically developed for binary detection, a rather difficult subject given the low-level features of binary code, this deep learning model, Vulanalyzer, accurately preserves instruction semantics and structural linkages by using sequential and topological learning to mimic program execution through recurrent units and graph convolution, especially in assembly code [45]. Mostly distinguished by its multi-head attention system, which stresses pertinent commands and fundamental blocks, Vulanalyzer underlines fundamental directions and basic blocks. This encourages developers to concentrate on the code components the model considers most indicative of a vulnerability, therefore producing interpretable results. Although it clarifies things, Vulanalyzer, like GAVulExplainer, does not especially mix causal reasoning, causal inference, or causal deep learning (CDL). Emphasizing the need of greater study on causal approaches that can identify the why behind vulnerability projections, the emphasis stayed on simplifying difficult interactions and improving interpretability by means of attention processes. The writers mostly concentrated on the lack of explainability and the challenges to elucidate intricate links in binary code. With topological and sequential learning combined, the approach captures structural relationships as well as instructional meanings, so requiring minimal topic knowledge. Still, even if attention processes help to define the what of the model's judgments, they do not naturally define the *why* - the fundamental causal factors generating the sensitivity. Most importantly, the method can be applied over multiple architectures and code obfuscation methods. The research on the development of explainable functional summaries of assembly code performed by Taviss et al. [45] emphasizes understanding code semantics in vulnerability analysis.

Moschitti et al. [25] presented an XAI-based system for assessing computer code in a graph environment. This system evaluates the significance of syntactic structures for Common Weakness Enumeration (CWE) classification [24], therefore connecting learnt code feature representations to subtle semantics recognized by security professionals. The system generates ranks of syntactic constructive contribution levels in Abstract Syntactic Trees (AST) among CWE types for Java and C++ datasets. A novel feature-masking method, varying the neighbourhood of code tokens and syntactic constructions, is applied for the graph environment. The change in the code token neighbourhood is transformed into the CWE-type similarity score by means of information retrieval approaches. The authors showed how nuanced semantics understood by security professionals might be connected to the learnt code feature representations by CWE similarity generated from XAI explanations. This approach did not, however, particularly target causal reasoning or causal deep learning (CDL). The authors admitted that present XAI systems have several limits, namely, their incapacity to generalize to undiscovered vulnerability patterns and transcend the scope of input data. The constraints covered are those of interpretability of acquired features and transferability of learned patterns to different datasets. Although graph-based representations and ASTs are widely used in vulnerability identification, their efficacy may vary depending on the programming language and the particular vulnerabilities under attack. The research of Allamanis et al. [1] on machine learning for massive code and naturalness offers a larger background for appreciating the difficulties of expressing and evaluating code.

Hajipour et al. [22] developed a framework for vulnerability threat prediction. This method generated a semantic representation and computed an explainable threat score by prioritizing research activities using topic modeling of vulnerability descriptions (from sources like the National Vulnerability Database). Furthermore, included was a fresh trend score based on internet infosec conversations to pinpoint popular vulnerabilities. These results were aggregated on a visual dashboard to give investigative work top priority. The main contribution of the authors was the computation of a new trend score and an explainable threat score based on the topic model. This framework offers a semantic representation of vulnerabilities constructed using topic modeling of vulnerability descriptions. The framework offers a semantic representation of vulnerabilities constructed using topic modeling of vulnerability descriptions. Although helpful for prioritizing, it did not investigate the fundamental *causal* elements affecting vulnerability exploitability, like particular code patterns or interactions increasing the probability of the exploitation of a vulnerability. Knowing which weaknesses are most likely to be utilized against you will help you to allocate resources on the most crucial risks. Online discussions and vulnerability descriptions from other sources provide some of the framework's material. It may thus be biased and have restrictions on its timeliness and completeness.

Allix et al. [32] investigated the use of deep learning and explainability methods (most especially SHAP) toward this aim. Their results showed that explainability techniques might occasionally present distracting information, therefore harming rather than supporting engineers; code characteristics employed by DL models were often only partially connected to the underlying causes of vulnerabilities. This emphasized the need for more accurate localization and the incorporation of causal reasoning to uncover the true underlying causes. This allows for a deeper understanding that goes beyond mere correlations. The main contribution of the authors was to assess how well explainability and deep learning (DL) approaches could localize source code assertions concerning vulnerabilities. Using two deep learning (DL) techniques, VulDeePecker and JavaBert, the authors localized source code phrases pertaining to vulnerabilities using SHAP, a model-agnostic explainability method. The paper emphasizes that current explainability methods do not provide developers with practical insights effectively. Important determinants of the results of the study are the choice of deep learning models (VulDeePecker and JavaBert) and the respective application of SHAP. Practical implementations depend critically on more programming language-oriented encoding approaches recommended by Allix et al. [32] and the evolution of more advanced techniques for vulnerability localization.

2.3 Research Gaps

Despite substantial empirical success achieved by deep learning (DL) approaches in software vulnerability detection, several fundamental limitations persist that compromise the reliability, generalization, and actionable utility of these systems in industrial settings. These deficiencies collectively define the necessity for the proposed research framework.

2.3.1 The Causal Deficiency and Generalization Gap

The first major challenge stems from the inherent reliance on statistical correlations over genuine causal mechanisms in program analysis. Current detectors frequently learn surface-level regularities or spurious correlations that co-occur with vulnerabilities in training data. This methodological bias compromises model robustness, manifesting as fragility under benign code transformations and significant performance degradation during cross-project domain shift. This brittleness suggests that existing normalization or domain-adaptation strategies only partially mitigate the underlying causal deficiency, leading to unstable predictions and poor transferability in dynamic, evolving code repositories. Consequently, many models fail to capture the true generative flaw mechanism.

2.3.2 The Interprocedural Reasoning Gap

The second critical deficiency is the limited capability for interprocedural reasoning and mechanism reconstruction. The majority of current learning frameworks are constrained to analyzing function-level slices or local subgraphs. This constraint prevents them from effectively tracing the end-to-end vulnerability lifecycle: the root cause → propagation path → exploitable sink. Without a mechanism-aware representation that explicitly models data and control dependencies across function boundaries, detectors default to highlighting local saliency, failing to reconstruct the comprehensive, executable narrative required by developers for complete, verifiable remediation.

2.3.3 The Causal Evaluation Gap

A third limitation concerns evaluation. Standard metrics such as Accuracy, Precision, Recall, F1, and mAP quantify label agreement but do not assess whether a model's attributions are causally faithful or stable under principled counterfactual edits. In the absence of standardized *causal* metrics, it remains difficult to determine whether a detector has learned the generative mechanism of a vulnerability or merely a pattern correlated with it.

To address these gaps, a novel framework is required, one that integrates causal reasoning, contextualization, and attention-driven graph-based learning. Such a system must be capable of identifying spurious correlations and performing a counterfactual analysis. It should construct contextual causal chains that trace how vulnerabilities propagate across components, monitor causal evaluation metrics to ensure transparency, and leverage graph-based neural networks to effectively capture interprocedural dependencies. In addition, it should implement context-sensitive attention mechanisms to prioritize the most security-relevant code segments.

2.4 Research Challenges and Rationale

The field of automated vulnerability analysis has progressed considerably over the past few years, yet several fundamental challenges continue to limit both reliability and interpretability. A closer reading of the literature reveals that many reported improvements in predictive accuracy obscure issues that have practical consequences for real-world deployment. Publicly available corpora often contain near-duplicate samples, mislabeled or weakly supervised instances, and severe class imbalance, all of which inflate apparent performance gains and make reproducibility difficult to achieve [42, 6]. Moreover, a substantial portion of existing datasets are trimmed to function-level snippets that conceal the cross-function control and data flows from which actual exploits emerge. While newer datasets have made progress in improving curation and granularity, comprehensive resources remain rare. The *ReposVul* dataset represents a notable advancement by preserving repository-level context with multi-granular dependencies and explicit call relations [48]; however, the research community still lacks standardized multi-factor benchmarks capable of jointly evaluating accuracy, robustness to benign code transformations, interprocedural coverage, and explanation quality across root, propagation, and sink components. The absence of such benchmarks makes it difficult to measure generalization beyond narrow or synthetic settings, thereby slowing the transition of academic progress into practice.

Representation remains a pivotal element in achieving reliable and interpretable vulnerability detection. Sequence-based models, including transformer architectures, perform competitively in benchmark settings but struggle to capture control and data dependencies that underlie real exploitability [12, 6]. In contrast, graph-based representations such as Abstract Syntax Trees (AST) for syntactic structure, Control-Flow Graphs (CFG) for execution order, Data-Flow Graphs (DFG) for variable influence, Program Dependence Graphs (PDG) for joint control–data dependence, and Code Property Graphs (CPG) for unified representations [28], have shown promise in encoding the structural context critical to vulnerability comprehension. Yet, empirical performance remains highly sensitive to how interprocedural semantics are represented—specifically, the inclusion of call edges, argument→parameter and return→caller bindings, and approximations for aliasing or points-to relationships. Structure-aware pretraining frameworks such as *GraphCodeBERT* have improved downstream generalization by incorporating dataflow signals into token-level embeddings [15, 42]. However, the extent of this improvement depends heavily on granularity, dataset diversity, and the quality of interprocedural signal integration.

Beyond static representation, the process of assembling long, executable vulnerability paths from root cause to sink presents an additional challenge that is both combinatorial and semantic. Naive best-first or depth-first search strategies tend to drift into locally plausible but globally inconsistent chains, while global optimization strategies such as Integer Linear Programming (ILP) or Satisfiability Modulo Theory (SMT) are computationally intractable at practical scales. This exposes a structural gap in current systems, where many either perform unconstrained search, leading to instability and irreproducibility, or depend on handcrafted heuristics that fail to capture the genuine progression of interprocedural flaw propagation. The research presented here addresses this issue by employing a *constrained beam search* procedure enhanced with an *Adaptive Causal Contextualization (ACC)* layer that enforces structural constraints including control- and data-flow reachability, interprocedural legality, and cycle avoidance. Additionally, a compact *Causal Knowledge Graph (CKG)* is mined from training data to serve as a small but informative prior over relation n-grams. This CKG encodes directional motifs derived from empirical code patterns, gently biasing the decoding process toward realistic causal transitions without overriding correctness validations performed by ACC. The combination of structured constraints and empirical causal priors reduces search entropy and improves the stability and coherency of long interprocedural traces.

Another enduring barrier to adoption is model brittleness under realistic development environments. A wide range of architectures spanning LSTMs, graph neural networks (GNNs), and transformers have demonstrated significant degradation when transferred across projects, programming styles, or naming conventions [42]. Even minor, semantics preserving edits can cause large fluctuations in predictions.

While model adaptation strategies mitigate these effects to a degree, most existing pipelines remain correlation driven and fail to develop explicit awareness of causal mechanisms [30]. Furthermore, a persistent simplifying assumption in the literature is that vulnerabilities exist within isolated functions, leading to overly optimistic performance estimates that overlook cross-component propagation and realistic exploit surfaces [41]. Post-hoc explanation techniques, including attention visualization and perturbation analysis, have been widely applied, but these often highlight spurious or overly localized features that do not align with the actionable causes developers must address [32, 25].

Recent research trends have increasingly turned toward causality-aware learning, encompassing counterfactual training formulations, dual factual–counterfactual evaluation strategies, and graph-based editing techniques designed to test causal stability [3, 9, 39, 21, 4]. While these methods offer improvements in robustness and interpretability, the field still lacks consensus metrics for quantifying causal fidelity specifically, the stability of model predictions under targeted interventions and the alignment of internal attributions with ground-truth causal elements. These limitations define the core motivation for this research.

The work undertaken in this thesis specifically addresses these gaps by focusing on interprocedural vulnerabilities that propagate across functions, files, and modules. Programs are encoded as enriched Code Property Graphs with explicit cross-boundary semantics, enabling faithful modeling of causal dependencies. Using *constrained beam search* guided by *Adaptive Causal Contextualization (ACC)*, and assisted by a lightweight *Causal Knowledge Graph (CKG)* prior derived from frequent relational motifs, the proposed framework explicitly constructs a single, executable root$\rightarrow \cdots \rightarrow$sink causal chain for each detected vulnerability. This approach ensures both structural legality and semantic plausibility throughout the reasoning process. To determine whether the model captures underlying mechanisms rather than dataset-specific correlations, conventional detection metrics are complemented by causal criteria that evaluate predictive stability under intervention (counterfactual consistency) and structural attribution alignment. Collectively, these components yield a chain-centric, interprocedural, and causally principled methodology aimed at producing transparent, auditable explanations that developers can meaningfully interpret, validate, and act upon.

Chapter 3

Methodology

This chapter presents the complete learning pipeline developed for this thesis, designed to achieve two complementary objectives: accurate vulnerability detection and faithful reconstruction of executable interprocedural causal chains. The overarching goal is not merely to identify code fragments that are susceptible to exploitation but to connect each vulnerability's root cause to its propagation and eventual sink through semantically meaningful, executable paths. This approach directly addresses the interpretability and robustness limitations of existing deep learning models, which often identify vulnerable statements in isolation without reconstructing their broader causal mechanisms.

The proposed methodology builds upon an augmented Code Property Graph (CPG) that unifies core program representations, including the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG), along with interprocedural elements such as call graphs, argument-to-parameter bindings, return-to-caller mappings, and alias analyses. This integrated graph structure provides a rich substrate capable of capturing both lexical and structural properties of source code while maintaining execution consistency across function boundaries. It enables the representation of full vulnerability mechanisms that interlink syntactic context, control dependencies, and data propagation chains within and across program scopes.

At the foundation of this approach lies a three-stage architecture that systematically transforms raw code into semantically enriched and causally interpretable representations. The first stage, structure-aware feature initialization, employs GraphCodeBERT to encode both lexical and data-flow attributes of code tokens. Through large-scale pretraining on diverse software repositories, GraphCodeBERT produces context-sensitive embeddings that capture syntactic roles and semantic dependencies, thereby establishing strong inductive priors for vulnerability representation and downstream learning [15].

In the second stage, a graph attention encoder aggregates contextual information from the heterogeneous CPG. Multi-head attention mechanisms systematically combine neighborhood features, allowing the network to emphasize relationships supported by executable semantics such as def-use relations and control dependencies. To ensure interpretability and robustness, the encoder is regularized using causality-oriented constraints that guide attention toward plausible execution paths while discouraging reliance on spurious correlations. This strategy is inspired by causality-aware models that have demonstrated improved generalization by aligning learning objectives with causal structure consistency [3, 39, 21].

The final stage introduces Adaptive Causal Contextualization (ACC), the component responsible for assembling an executable causal chain from the attention-guided graph representation. ACC formulates vulnerability mechanism recovery as a constrained path search problem, selecting and linking nodes that represent the root cause, its propagators, and the exploitable sink. It enforces interprocedural feasibility through control and data-flow constraints, validity across call-return relations, and parsimony in produced chains. The outcome is a single, interpretable, and verifiable causal chain consistent with real program execution semantics.

Model training combines standard classification objectives for vulnerability detection with causal regularization terms that penalize incoherent or non-executable attention patterns. The overall learning framework promotes representations that remain stable under benign code refactorings and generalize across projects and programming paradigms. It draws conceptual inspiration from recent advances in structure-aware vulnerability detection and causal representation learning, integrating their strengths into a unified, chain-centric architecture [28, 42, 4, 9, 17].

Through this structured pipeline, the proposed method advances beyond point-level vulnerability classification toward executable, chain-level causal inference. The chapters that follow detail the architecture's constituent stages, including formal definitions, optimization design, and algorithmic implementation, supported by empirical evidence demonstrating its precision, interpretability, and robustness in real software systems.

3.1 Chain-Centric Program Representation

3.1.1 Dataset Description, Ground Truth Formation, and Preprocessing Pipeline

This chapter documents the corpus adopted for experimentation, the ground-truth definition used to label vulnerable code, and the full preprocessing pipeline that converts repositories into chain-centric graphs suitable for interprocedural analysis. I use *ReposVul*, a repository-level dataset designed to untangle patches, expose multi-granularity dependencies, and filter outdated fixes, which aligns with interprocedural, chain-centric evaluation [48].

3.1.2 ReposVul: scope and characteristics

ReposVul integrates and links vulnerability metadata from CVE and CWE databases with detailed patch commit information and source code snapshots both before and after the application of each fix. A typical dataset entry couples CVE identification, weakness categorization, and vulnerability severity metrics with associated patch commit metadata and the code diffs representing vulnerable and fixed code versions. Table 3.1 summarizes these core data attributes as reported in the original corpus documentation.

Table 3.1: Key metadata fields per entry in ReposVul

Category	**Representative Fields**
Vulnerability Entry	CVE-ID, CWE-ID, language, external references, CVE description, publishing date, CVSS vector and its components (AV, AC, PR, UI, S, C, I, A)
Patch Metadata	Commit-ID, commit message, commit date, project and repository identifiers, parent and child commit links, API and web URLs
Related Files	File name, programming language, vulnerable and fixed code versions, line-level diffs, URL of the source file

The dataset construction procedure involves four major stages. Firstly, raw data crawling collects vulnerability reports and patches from public databases and major code forges, along with full repositories at the parent and child commit states to reconstruct complete pre- and post-fix source code snapshots. Secondly, vulnerability untangling applies a joint decision rule that combines large language model (LLM) assessments of code change relevance with static analysis heuristics to identify and isolate files genuinely involved in vulnerability fixes, excluding unrelated refactorings or additional feature changes. Thirdly, multi-granularity dependency extraction mines caller–callee relationships across the entirety of each repository, expanding beyond direct diffs to incorporate top-level functions and inter-file connectivity, vital for capturing interprocedural vulnerability mechanisms. Finally, trace-based filtering discards outdated or superseded patches using commit chronology and file path tracking, ensuring the dataset contains only up-to-date and correct vulnerability fixes [48].

The objective of this thesis is to reconstruct executable interprocedural chains. ReposVul provides repository-level context with parent and child patches, multi-granularity slices at line, function, file, and repository levels, and caller–callee relationships mined across the repository [48]. This combination enables chain-centric ground truth and realistic evaluation of cross-boundary propagation.

3.1.3 Extraction and graph construction pipeline

The conversion from raw dataset entries to chain-centric program graphs occurs over six sequential stages. The first three follow the dataset's framework to yield high-quality, curated examples; the latter three transform these examples into program graphs amenable to interprocedural causal analysis.

Stage A: raw vulnerability and patch acquisition. I ingest the ReposVul entries with their CVE and CWE metadata and fetch, for each entry, the project repository state at the parent and child commits. This guarantees access to the full files before and after the fix. Commit identifiers, messages and dates, project identifiers, and file paths for every changed file are indexed for traceability.

Stage B: vulnerability untangling. Patches frequently combine refactorings with fixes. I apply the dataset's untangling procedure, which combines model judgments on code-change, to retain only files judged vulnerability-fixing related by the joint rule. Both signals are persisted for audit.

Stage C: multi-granularity dependency extraction. For each retained patch, I extract the caller–callee relationships necessary to connect a candidate source to a sink via interprocedural flows. Repository-level extraction provides cross-file call links, and the search scope is expanded to top-level functions when needed so that call chains outside the direct diff are captured.

Stage D: code normalization and static graph building. I reconstruct a heterogeneous program multigraph per repository snapshot that merges the Abstract Syntax Tree (AST), the Control-Flow Graph (CFG), the Data-Flow Graph (DFG), and interprocedural call and return links. Argument→parameter and return→caller bindings are attached where resolvable, and conservative points-to based alias links are included to approximate def–use through pointers and containers. Node features encode token information, syntactic category, type hints, and lexical normalization flags. Edge features encode relation type and direction. Normalization anonymizes identifiers, buckets literals, and removes formatting that does not carry semantics, which reduces variance from benign refactorings reported to affect learning-based detectors [6, 42].

Stage E: patch-aware differencing and slice materialization. I compute precise line changes for each patch, then materialize four synchronized views for every example: line-level edits, enclosing functions, touched files, and a connected repository-scope subgraph that reaches from the touched region to any reachable sink along feasible control- and data-flow paths. These synchronized views instantiate the dataset's multi-granularity design in a graph-native form and supply the raw material for chain assembly.

Stage F: trace-based filtering for outdated patches. I reapply the dataset's trace-based filter. File paths and commit times are tracked to remove nonfunctional files, then changed files across parent and child commits are compared to identify outdated patches superseded by subsequent fixes to the same file. Eliminating such cases avoids training and testing on incomplete or obsolete fixes and improves the validity of interprocedural chains that depend on correct call and data-flow context.

Table 3.2: Preparation pipeline summary.

Stage	Key operations and outputs
A	Ingest CVE, CWE, and patch metadata; fetch pre- and post-fix files; index commits and file paths.
B	Apply joint untangling to retain vulnerability-fixing files.
C	Extract repository-level caller–callee links; expand to top-level functions when needed.
D	Build AST, CFG, and DFG with call and return links; attach argument→parameter, return→caller, and alias edges; type nodes and edges; normalize code.
E	Compute line deltas; assemble synchronized line, function, file, and repository subgraphs that preserve feasible paths to sinks.
F	Remove outdated patches using path and commit chronology; keep only current fixes.

3.1.4 Unified multigraph, typing, features, and storage

The resulting representation is a unified heterogeneous multigraph rooted in the CPG abstraction [55]. Within a single typed node store, I maintain relation-specific edge sets for AST, CFG, and DFG, together with explicit interprocedural links for *CALL* (caller to callee entry), *ARG2PARAM* (actual→formal), *RET2CALL* (callee return→call site), and *RET2LHS* (return value→assignment target), plus reverse edges for all relation families to support undirected neighborhood aggregation during learning. This topology ensures that control and data dependences, as well as cross-boundary value flows, are simultaneously available for chain reconstruction.

Nodes are typed by syntactic role (for example identifier, literal, operator, statement, basic block, function) and carry a compact structural feature vector that includes token category flags, simplified SSA indices where available, type hints, and normalization indicators. Edges are typed by relation family and direction, and optionally include guard predicates for CFG edges or alias provenance for points-to–induced links. Graphs are persisted in shard files with per-graph metadata such as repository, commit, file path, and function names, which makes the pipeline scalable and allows constant-time retrieval of instance-level provenance.

To accommodate later feature enrichment, I store two parallel encodings. The *base* encoding contains compact structural features adequate for ablations and classical GNNs. The *pretrained* encoding augments each node with a 768-dimensional structure-aware embedding derived from a data-flow–aware encoder (GraphCodeBERT), yielding a dual-channel representation: a low-dimensional structural feature vector and a high-dimensional contextual vector. This dual-channel design is consumed by the model, and it preserves a consistent topology across the base and enriched variants so that interprocedural edges and counts remain unchanged while only the feature space is expanded.

For decoding I treat the set of relation types that appear in the heterogeneous graph (AST, CFG, DFG, CALL, ARG2PARAM, RET2CALL, RET2LHS, and DFG_-THIN) as a fixed *alphabet* $\mathcal{R}$. This same alphabet underlies the causal prior introduced later: the prior never changes the graph; it only provides small, auditable preferences over which relation type to expand at each hop during decoding.

3.1.5 Ground-truth definition

Ground truth is defined at two complementary levels. At the label level, a repository revision is marked vulnerable if it appears in a vulnerability-fixing commit pair documented by ReposVul, and the corresponding fixed revision is marked non-vulnerable. Using positive–negative pairs with identical surrounding context reduces confounding from unrelated changes [48]. Commits that aggregate multiple, unrelated fixes are excluded. At the mechanism level, each example carries a single executable chain that reflects the vulnerability mechanism and is represented as an ordered sequence of nodes and edges from a source through any propagators to a sink, possibly crossing function boundaries. Sources are origin sites where attacker-controlled or otherwise untrusted data enters the program state, for example input acquisition or deserialization. Sanitizers are code regions that validate, constrain, or transform tainted data, for example bounds checks, format validation, or defensive copies. Propagators are statements or calls that forward tainted or unsafe state across assignments, pointer dereferences, parameter passing, returns, container writes, and index arithmetic. Sinks are security-relevant operations that become exploitable once reached by tainted or unsafe state, for example memory writes or indexing, command execution, path traversal, database execution, or dangerous casts. Role assignment is automated with a rule base derived from secure-coding literature and prior datasets, then verified with control- and data-dependence consistency checks. A function-name lexicon and API families capture common sources and sinks, and structural patterns capture sanitizers and propagators. Appendix D lists the lexicon and patterns.

The annotation workflow begins by seeding candidates from diffs, since line edits often expose guard predicates and argument shaping. Def–use traversal then collects propagators and checks reachability to known sink patterns. Calls and returns are resolved along the repository-level call graph, and argument→parameter as well as return→caller bindings are attached at call sites. Consistency checks require that a feasible path exists from every retained source to a sink, that each sanitizer blocks at least one tainted path, and that each propagator lies on a feasible CFG-consistent path. Two verification passes improve quality. A feasibility pass ensures that removing the sink breaks exploitability in the slice and that the path conditions are satisfiable under conservative approximations. A counterfactual plausibility pass applies simple edits, for example strengthening a bounds check or swapping a dangerous sink for a benign variant, and re-checks reachability. Examples that fail are corrected or excluded.

3.1.6 Interprocedural Semantics and Cross-Boundary Validity

Each program graph instance integrates comprehensive interprocedural relationships, encompassing a full repository-level call graph enriched with call-site contextual information and parameter arities. Edges explicitly bind actual arguments to formal parameters and return values to caller variables. Library API summaries capture known taint behavior conservatively. Points-to based alias edges approximate data propagation through pointers, references, and container structures. Every interprocedural edge records provenance data, including the source file and enclosing function of both endpoints, facilitating auditability and later visualization in case studies.

Empirical evidence supports the validity of this interprocedural modeling for vulnerability chain reconstruction. The repository-level dependency extraction, coupled with scope expansion beyond syntactic diffs, recovers call chains that cross files as needed to connect vulnerability roots to corresponding sinks. As shown in Table 3.5, a significant fraction of instances contain non-empty caller or callee sets, with a consistent subset exhibiting both, indicating traversable cross-function edges. The feasibility and counterfactual plausibility checks include the interprocedural paths, thus ensuring that the retained chains represent executable and causally coherent vulnerability mechanisms rather than disconnected local saliencies.

3.1.7 Data Splits, Controls, and Reproducibility

ReposVul provides official, project-level train, validation, and test splits specifically designed to measure cross-project generalization and to prevent data contamination via code duplication or patch ancestry leakage. This research adopts these official splits without modification to maintain compatibility and benchmark integrity [48].

All files from any single project remain confined to the same partition, and when robustness over time is evaluated, the splits respect commit chronology.

Strict controls enforce the following: parent and child patch commits never split across partitions, no identical file snapshot exists simultaneously in training and test sets, identical CVE vulnerabilities do not appear in multiple splits, and call graph artifacts spanning several repositories are not merged across splits. To support detailed evaluation and reproducibility, each example records the ordered node ID sequence constituting the causal chain, including assigned node roles, involved edge types, and synchronized line, function, and file indices. Random seeds are fixed for all sampling operations, parsers and language versions are logged, and the commit hashes of all repositories are tracked. Open publication of preprocessing configuration, graph counts, split checksums, and filtering statistics further facilitates replicability.

3.1.8 Empirical Coverage and Dataset Statistics

Tables 3.3–3.5 report split-wise statistics for the prepared C and C++ subset used in my experiments, including label balance, graph-instance density, and interprocedural connectivity.

Table 3.3: Split-wise label counts at the file-level snapshot granularity.

Split	Records	Non-vuln	Vuln	Pos.%
Train	185,791	180,259	5,532	2.98
Valid	23,224	22,503	721	3.10
Test	23,224	22,554	670	2.88

Table 3.4: Graph instances and node-level label density after chain centric conversion.

Split	Graphs	Vuln nodes	Non-vuln nodes	Pos. ratio
Train	3,438	9,946	25,173,258	3.95×10^{-4}
Valid	2,905	1,455	3,970,281	3.66×10^{-4}
Test	2,915	1,316	3,973,974	3.31×10^{-4}

Table 3.5: Interprocedural connectivity of prepared examples: presence of non-empty caller and callee sets.

Split	Caller%	Callee%	Both%	Caller_chg%	Callee_chg%	Both_chg%
Train	12.59	28.95	8.72	0.46	2.79	0.06
Valid	13.10	29.26	9.21	0.55	2.74	0.07
Test	12.97	29.17	9.03	0.47	2.90	0.09

These figures indicate that nearly one third of instances expose a callee set, about one eighth expose a caller set, and roughly one in ten expose both, which together provide the minimum structural precondition for discovering interprocedural chains that cross function boundaries. Because the executable chains are validated with feasibility and counterfactual checks in the presence of these links, the prepared data sustain interprocedural vulnerability quality rather than relying on isolated, intra-procedural signatures.

3.1.9 Summary

The described dataset preparation pipeline produces a suite of interprocedurally rich, chain-prepared program graphs annotated with explicit source, sanitizer, propagator, and sink roles that reflect executable paths rather than isolated statements. ReposVul provides reliable label provenance and repository context [48], and the adoption of its official splits together with normalization and verification reduces known threats to validity in learning-based vulnerability detection [6, 42]. This facilitates the chain-centric learning and evaluation paradigm advanced in this thesis. The choice of ReposVul ensures reliable label provenance, broad repository context, and integration of multi-level dependency information. By adopting the official splits and applying thorough normalization and verification, the dataset mitigates prevailing threats to learning validity as reported in prior literature. Collectively, the resulting dataset and methodology establish a strong foundational platform for the methodology and experimental chapters that follow.

3.2 Model Architecture

3.2.1 GraphCodeBERT Feature Initialization

This section details how I initialize node features with *GraphCodeBERT* [15] and fuse them with the structural descriptors extracted from the chain-centric program graphs. GraphCodeBERT is a transformer encoder pretrained on code with both masked language modeling and a data–flow aware objective, which links tokens that participate in definition–use relations. Because the program graphs already expose explicit data flow, call and return links, and argument→parameter as well as return→caller bindings, this initialization provides a complementary token-level view that is sensitive to the same dependencies that drive causal chain assembly. Prior empirical work reports that structure-aware pretraining can improve cross-project robustness when combined with graph features, although the magnitude of gains depends on task granularity and project diversity [42].

Input construction and tokenization: For every graph node with a concrete source span, I recover the normalized code fragment produced during preprocessing, then tokenize it using the official GraphCodeBERT byte-pair tokenizer (vocabulary ~50k, maximum length L=512). Long spans are chunked into overlapping windows of length L and stride S (default S=384), after which window-level embeddings are aggregated back to the node. Normalization anonymizes identifiers and buckets literals while preserving types and positions, which reduces variance due to benign refactorings without erasing the data–flow cues exploited by the model [15, 42].

Token–node alignment: Let $\{t_1, \ldots, t_T\}$ be the token sequence for a fragment, and let $\mathbf{E} \in \mathbb{R}^{T \times d_t}$ be the corresponding matrix of contextual token embeddings from GraphCodeBERT with hidden size d_t=768. Each node v_i maps to a set of token indices $\mathcal{T}(i) \subseteq \{1, \ldots, T\}$ based on its character span. I compute a simple mean to obtain the node's textual embedding

$$\tilde{\mathbf{x}}_i^{\text{text}} = \frac{1}{|\mathcal{T}(i)|} \sum_{t \in \mathcal{T}(i)} \mathbf{E}_t \in \mathbb{R}^{768}. \tag{3.1}$$

Equation: (3.1) averages the GraphCodeBERT vectors of all subword tokens that realize node v_i in text, producing a single 768-dimensional representation for that node.

Edge-aware pooling for statement nodes: For nodes that correspond to full statements containing several identifiers, I apply a light attention that prefers tokens which carry definition–use signal. Let $\phi(t) \in \mathbb{R}^k$ denote a small vector of local data–flow attributes for token t, for example definition or use role and fan-in or fan-out counts. I compute

$$\alpha_{i,t} = \frac{\exp\big(\mathbf{u}^\top \tanh\big(\mathbf{W}_e[\,\mathbf{E}_t \,\|\, \phi(t)\,]\big)\big)}{\sum_{s \in \mathcal{T}(i)} \exp\big(\mathbf{u}^\top \tanh\big(\mathbf{W}_e[\,\mathbf{E}_s \,\|\, \phi(s)\,]\big)\big)}, \tag{3.2}$$

$$\mathbf{x}_i^{\text{text}} = \sum_{t \in \mathcal{T}(i)} \alpha_{i,t}\, \mathbf{E}_t, \tag{3.3}$$

where $\mathbf{W}_e$ and $\mathbf{u}$ are learned parameters and $[\,\cdot \,\|\, \cdot\,]$ denotes concatenation. If $\mathcal{T}(i)$ contains no identifier tokens, I fall back to the uniform average in (3.1). **Explanation:** (3.2) assigns higher weights to tokens that look more like def–use carriers, then (3.3) aggregates them into one statement-level vector.

Data–flow conditioning inside the LM: When a node's span and its immediate def–use neighbors fit within the same token window, I pass those intra-window def–use edges to GraphCodeBERT so that its pretrained data–flow attention mask is active [15]. Cross-window and interprocedural flows are modeled explicitly by the graph encoder via DFG and call or return edges.

Fusion with structural descriptors: Independently of text, every node v_i has a structural feature vector $\mathbf{x}_i^{\text{struct}} \in \mathbb{R}^{d_s}$, which encodes node type, role indicators, degree buckets, simple SSA indices when available, and literal statistics. I project both textual and structural branches to a common hidden size d_0, then combine them with a learned gate:

$$\mathbf{h}_i^{\text{text}} = \text{LN}\big(\mathbf{W}_t\, \mathbf{x}_i^{\text{text}}\big), \qquad \mathbf{h}_i^{\text{struct}} = \text{LN}\big(\mathbf{W}_s\, \mathbf{x}_i^{\text{struct}}\big), \tag{3.4}$$

$$g_i = \sigma\Big(\mathbf{w}_g^{\top}\big[\,\mathbf{h}_i^{\text{text}} \,\|\, \mathbf{h}_i^{\text{struct}}\,\big] + b_g\Big), \tag{3.5}$$

$$\mathbf{h}_i^{(0)} = g_i\,\mathbf{h}_i^{\text{text}} + (1-g_i)\,\mathbf{h}_i^{\text{struct}} \in \mathbb{R}^{d_0}, \tag{3.6}$$

where $\mathbf{W}_t \in \mathbb{R}^{d_0 \times 768}$ and $\mathbf{W}_s \in \mathbb{R}^{d_0 \times d_s}$ are learned projections, LN is layer normalization, σ is the logistic function, and $g_i \in [0, 1]$ is a scalar gate. **Explanation:** (3.4) maps text and structure into the same space, (3.5) computes how much to trust text relative to structure, and (3.6) forms the final node initialization given to the graph encoder.

Long-span and cross-window aggregation: If a node's span appears in multiple token windows, I first apply (3.1) or (3.3) within each window, then average the resulting vectors across windows. For function-proxy nodes that summarize whole bodies, I pool their child statement vectors to create a coarse function representation while keeping statement nodes localized.

Freezing, partial fine-tuning, and caching: To attribute improvements primarily to the chain-centric encoder rather than full-model adaptation, I freeze GraphCodeBERT by default. In an ablation, I unfreeze the top k transformer layers (typically k=2) with a reduced learning rate η_{LM}=$\eta_{\text{GNN}}/10$ and gradient clipping at 1.0, then report both regimes [42]. Token hidden states are precomputed offline and cached as FP16 tensors keyed by repository, commit hash, file path, and character span, with a manifest that records checkpoint identifier, tokenizer version, maximum length, stride, and normalization settings for reproducibility.

Dimensional summary: Table 3.6 summarizes the principal dimensions. The fused size d_0 equals the width of the first graph-attention layer. Relation types for edges, for example AST, CFG, DFG, CALL, ARG2PARAM, RET2CALLER, are separately embedded as small vectors and used later in relation-aware attention.

Table 3.6: Node feature dimensions at initialization.

Component	Dim.	Notes
Textual embedding	768	GraphCodeBERT contextual vector [15]
Structural raw	100–200	Type, role, degree, SSA hint, literal buckets
Projected text	d_0	$W_t : \mathbb{R}^{768} \to \mathbb{R}^{d_0}$, with layer norm
Projected structural	d_0	$W_s : \mathbb{R}^{d_s} \to \mathbb{R}^{d_0}$, with layer norm
Fused node init $h^{(0)}$	d_0	Gated combination for the GAT encoder

Computational profile: Embedding extraction is performed once, then reused for training. With L=512 and S=384, a single modern GPU processes on the order of 10^5 tokens per second, after which graph-batched training proceeds at the speed of the encoder. This separation keeps GPU memory predictable, since batching is by number of nodes and edges rather than by token length.

GraphCodeBERT supplies a data–flow aware language-model view that complements explicit structural features in the chain-centric graphs. The gated fusion in (3.6) yields stable and reproducible node initializations for the causality-oriented graph encoder, and the design choices above, for example freezing versus partial fine-tuning and attention over def–use tokens, are chosen to balance robustness and performance [15, 42].

3.2.2 GAT with Causality-Oriented Attention

This encoder consumes the fused node initializations $h_v^{(0)}$ from §3.2.1 and produces: (i) contextual node states for classification, (ii) a *seed* score used to automatically choose chain starting points, and (iii) a learned edge compatibility that guides chain assembly. Training and inference support *multiple roots* per graph without manual seeding: the model ranks seeds and launches beams from the top candidates.

Graph and notation: Let $\mathcal{G} = (\mathcal{V}, \{\mathcal{E}_r\}_{r\in\mathcal{R}})$ be the heterogeneous program graph. Nodes $v \in \mathcal{V}$ correspond to statements, expressions, or proxies. Relations $r \in \mathcal{R}$ include `AST`, `CFG`, `DFG`, `CALL`, `ARG2PARAM`, `RET2CALL`, `RET2LHS`, and a light `DFG_THIN` summary used for long hops. Hidden width is d_0 and I stack L layers (default d_0=64, L=3).

Relation-aware GAT update: Each layer aggregates relation-typed messages with attention weights that depend on both endpoints and the relation:

$$\alpha_{u\to v}^{(r,\ell)} = \mathrm{softmax}_{u\in\mathcal{N}_r(v)}\big(\mathrm{LeakyReLU}\big(a_r^\top[\,W_r^{(\ell)}h_u^{(\ell)} \,\|\, W_0^{(\ell)}h_v^{(\ell)}\,]\big)\big), \tag{3.7}$$

$$h_v^{(\ell+1)} = \mathrm{ELU}\left(W_{\mathrm{self}}^{(\ell)}h_v^{(\ell)} + \sum_{r\in\mathcal{R}}\sum_{u\in\mathcal{N}_r(v)} \alpha_{u\to v}^{(r,\ell)}\, W_r^{(\ell)}h_u^{(\ell)}\right). \tag{3.8}$$

Here $h_v^{(\ell)} \in \mathbb{R}^{d_0}$ is the node state at layer ℓ, $\mathcal{N}_r(v)$ are the r-neighbors of v, $W_{\text{self}}^{(\ell)}, W_0^{(\ell)}, W_r^{(\ell)} \in \mathbb{R}^{d_0 \times d_0}$ are learned projections, and $a_r \in \mathbb{R}^{2d_0}$ is a relation-specific attention vector. Equation (3.7) assigns a normalized importance to each incoming r-edge based on the transformed endpoints, and (3.8) forms the next-layer representation by summing relation-typed messages plus a self-connection. This follows Graph Attention Networks for the attention mechanism with typed extensions inspired by relational and heterogeneous variants [47, 40, 49].

Node and edge scoring: After L layers I produce a node vulnerability logit, a seed score, and a relation-gated edge compatibility:

$$z_v = w_{\text{node}}^\top h_v^{(L)} + b_{\text{node}}, \qquad s_v = w_{\text{seed}}^\top h_v^{(L)} + b_{\text{seed}}, \tag{3.9}$$

$$c_{u \to v}^{(r)} = h_u^{(L)\top} B_r \, h_v^{(L)} + \beta_r, \tag{3.10}$$

with $w_{\text{node}}, w_{\text{seed}} \in \mathbb{R}^{d_0}$, biases $b_{\text{node}}, b_{\text{seed}} \in \mathbb{R}$, and a bilinear form $B_r \in \mathbb{R}^{d_0 \times d_0}$ plus scalar prior β_r per relation. Equation (3.9) scores nodes, and (3.10) scores directed edges for chain continuation.

Automatic multi-root seeding: Beams start from model-chosen seeds. Given seed logits $\{s_v\}$ I form a seed set

$$\mathcal{S}_K = \text{TopK}\big(\{s_v : v \in \mathcal{V}\}, K\big), \tag{3.11}$$

where K is chosen adaptively per graph: during training K equals the number of labeled sources when available, otherwise a cap (default K=8); during inference K is the top-p quantile or a fixed cap, whichever is smaller. No manual roots are provided. Each $v_0 \in \mathcal{S}_K$ initializes one beam.

Beam expansion and path scoring: From a current beam node v_{t-1}, all outgoing edges $(v_{t-1} \to v_t) \in \cup_r \mathcal{E}_r$ are candidate expansions. I score a path $\pi = (v_0, \dots, v_T)$ by

$$S(\pi) = \log \sigma(s_{v_0}) \; + \; \sum_{t=1}^{T} \Big(\alpha \, \log \sigma(z_{v_t}) + (1-\alpha)\, c_{v_{t-1} \to v_t}^{(r_t)}\Big), \tag{3.12}$$

where $\sigma(\cdot)$ is the logistic function, r_t is the relation of the chosen edge, and $\alpha \in (0,1)$ balances node and edge evidence (default α=0.7). Beams keep the top B partial paths by $S(\cdot)$ (default B=24), expand up to H hops (default H=5), avoid revisits, and stop early when a sink is reached or when no admissible successor exists. This procedure yields a ranked list of candidate chains per graph. The scoring is additive in log space, which is standard for beam search on graphs and mirrors path-based reasoning in knowledge graphs [10, 52]. During training, if multiple ground-truth roots exist, beams start from all of them via (3.11); if some roots are unlabeled, the learned seeds still discover them.

Training signals tied to chains: The encoder outputs feed three losses. Briefly, (i) node classification uses focal BCE on z_v with class balancing, (ii) an edge participation BCE pushes $c_{u \to v}^{(r)}$ up on edges that lie on high scoring or labeled chains and down on randomly matched non-chain edges, and (iii) a path ranking margin encourages $S(\pi)$ to exceed the score of random walks of equal length. When multiple ground-truth chains are present, losses are aggregated over all chains so the model learns a distribution over roots rather than a single origin.

Practical configuration and stability: I use L=3 layers with d_0=64, ELU activations, dropout 0.1 on node states and on attention logits in (3.7), AdamW with learning rate 2×10^{-3} and weight decay 10^{-4}, gradient clipping at 1.0, batch size selected so that a batch fits in GPU memory, and relation sets $\mathcal{R}$ that include both precise and summarized data-flow. Beam parameters are K=8, B=24, H=5, and α=0.7. In ablations I remove `DFG_THIN` to verify that long-range propagation remains correct; results show small drops on interprocedural cases, confirming its utility.

Hardware and software: Experiments run with PyTorch 2.4.1, CUDA runtime 12.1, one CUDA device visible, and an NVIDIA GeForce RTX 4070 Laptop GPU. The encoder is mixed-precision compatible; attention and bilinear edge scoring are computed in FP16 where safe and in FP32 when accumulating path scores in (3.12). With cached GraphCodeBERT features, throughput is dominated by graph batching and beam expansion rather than language model inference.

The encoder provides a clear separation of roles. Equations (3.7)–(3.8) build heterogeneous, relation-aware context following established GAT principles [47] with typed relations [40, 49]. Equations (3.9)–(3.10) provide node and edge scores, combining standard linear heads with a bilinear decoder. Equations (3.11)–(3.12) operationalize automatic multi-root discovery and beam-scored chain construction in line with prior path-search practice [10, 52]. Together these components yield an encoder that discovers and scores executable causal chains without manual seeds.

3.2.3 Causal Knowledge Graph (CKG): Mining, Findings, and Prior for Decoding

I estimate a compact *Causal Knowledge Graph (CKG)* over the relation alphabet $\mathcal{R}$ from the *training* graphs only. The CKG summarizes how relation choices tend to follow one another along executable propagation chains and is used solely as a weak guidance term during decoding.

Mining setup: Counts are collected from admissible chain hops in the training split (3,438 graphs), then smoothed with add-ϵ and optional temperature. I keep unigram priors $\widehat{P}(r)$, bigrams $\widehat{P}(r_t \mid r_{t-1})$, and a top-$K$ list of trigrams for interpretability.

Key findings (edge priors and start/end usage): Table 3.7 reports counts and priors per relation; Table 3.8 shows how chains typically start and end. Priors are sharply concentrated on CFG ($\approx$ 0.474) and CALL ($\approx$ 0.390). ARG2PARAM carries non-trivial mass ($\approx$ 0.091) and DFG_THIN is small but frequent enough to stabilize long hops ($\approx$ 0.039). RET2CALL and especially RET2LHS are rare in the mined chains.

Table 3.7: CKG edge priors from training graphs (counts and probabilities).

Relation	**Count**	**Prior**
CFG	12,970	0.4744
CALL	10,670	0.3903
ARG2PARAM	2,478	0.0906
DFG_THIN	1,055	0.0386
DFG	114	0.0042
RET2CALL	52	0.0019
RET2LHS	1	≈ 0

Table 3.8: Start/end relation frequencies observed in mined chains.

Relation	**Start count**	**End count**
CFG	6,494	1,965
CALL	170	3,612
ARG2PARAM	50	617
DFG_THIN	60	611
DFG	11	27
RET2CALL	50	2
RET2LHS	0	1

Key findings (bigrams and typical length): Bigrams show CALL→CALL is extremely self-persistent (≈ 0.999), CFG→CFG is common (≈ 0.583), CFG→CALL occurs with probability ≈ 0.309, and ARG2PARAM→ARG2PARAM is strong (≈ 0.743) with a notable branch to DFG_THIN (≈ 0.257). DFG_THIN shows self-loops (≈ 0.664) and transitions to ARG2PARAM (≈ 0.221). A simple hop histogram indicates 4-hop chains dominate among mined paths (6,835 instances), which aligns with an interprocedural root$\rightarrow \cdots \rightarrow$sink span.

Key findings (motifs): Top trigrams (Table 3.9) are dominated by CFG/CALL runs, plus interprocedural motifs involving ARG2PARAM and DFG_THIN. Notably, RET2CALL→CALL→CALL appears among the frequent motifs, supporting return-to-caller followed by continued call-graph traversal.

Table 3.9: Top mined trigram motifs (counts).

#	Count	Motif (tri-gram)
1	3,943	CFG→CFG→CFG
2	3,489	CALL→CALL→CALL
3	3,343	CFG→CALL→CALL
4	810	CFG→ARG2PARAM→ARG2PARAM
5	480	ARG2PARAM→ARG2PARAM→ARG2PARAM
6	348	ARG2PARAM→ARG2PARAM→DFG_THIN
7	288	CFG→CFG→CALL
8	180	CFG→CFG→ARG2PARAM
9	135	CFG→DFG_THIN→DFG_THIN
10	116	CFG→ARG2PARAM→DFG_THIN

Prior term (weak guidance only): During decoding, the model already assigns a score to each candidate chain. I add a small bonus to that score when the chain's edge sequence aligns with the CKG statistics (common unigrams, plausible bigrams, and a few named trigrams). with small nonnegative weights reflecting the observed dominance of bigrams and the usefulness of named trigrams for interpretability.

Updated beam score: Let $S(\pi)$ be the beam score from Eq. (3.12). I use

$$S^{\star}(\pi) \;=\; S(\pi) \;+\; \lambda\, S_{\text{CKG}}(\pi), \tag{3.13}$$

with λ=0.2 by default. The prior never changes admissibility; it only nudges ranking among admissible expansions.

3.2.4 Adaptive Causal Contextualization (ACC) for Chain Assembly

ACC turns the encoder outputs into a *single, executable, interprocedural* chain. It begins from model-selected seeds, expands only along admissible edges, and prefers paths whose internal roles and control structure match a plausible causal mechanism. All operations run on the heterogeneous graph constructed from the CPG backbone with interprocedural links [55].

Path state: A partial path $\pi = (v_0, \ldots, v_t)$ maintains four summaries: the current tip v_t; a taint footprint $\mathcal{T}(\pi)$ that aggregates reaching definitions and points-to classes; a call stack $C(\pi)$ that enforces well-nested call and return; and accumulated guards $\mathcal{G}(\pi)$ from the interprocedural CFG. These summaries allow constant-time checks while decoding.

Admissibility checks: From a tip $u = v_t$, a typed edge $u \xrightarrow{r} v$ is considered only if the following predicates all hold:

$$\mathsf{cfg\_ok}(u \to v) := \mathbf{1}\big[R_{\mathrm{CFG}}(u,v) = 1\big], \tag{3.14}$$

$$\mathsf{dfg\_ok}(u \xrightarrow{r} v, \mathcal{T}(\pi)) := \mathbf{1}\big[(r = \texttt{DFG}) \wedge \mathsf{tainted}(u, \mathcal{T}(\pi))\big] \;\vee\; \mathbf{1}\big[\exists g \in \mathcal{G}(\pi) : v \text{ is control dependent on } g\big], \tag{3.15}$$

$$\mathsf{ipa\_ok}(u \xrightarrow{r} v, C(\pi)) := \begin{cases} \text{push}(\text{callee}(v), \text{site}{=}u) & r = \texttt{CALL}, \\ \text{top of stack matches site of } u & r \in \{\texttt{RET2CALL}, \texttt{RET2LHS}\}, \\ 1 & \text{otherwise}, \end{cases} \tag{3.16}$$

$$\mathsf{alias\_ok}(u \xrightarrow{r} v, \mathcal{T}(\pi)) := \begin{cases} \mathbf{1}\big[\mathsf{pts}(u) \cap \mathsf{pts}(v) \neq \varnothing\big] & \text{if } r \text{ dereferences or writes}, \\ 1 & \text{otherwise}. \end{cases} \tag{3.17}$$

The overall admissibility is the logical conjunction

$$\mathsf{Adm}(u \xrightarrow{r} v \mid \pi) := \mathsf{cfg\_ok} \cdot \mathsf{dfg\_ok} \cdot \mathsf{ipa\_ok} \cdot \mathsf{alias\_ok}. \tag{3.18}$$

Equation (3.14) uses a precomputed interprocedural CFG reachability bit. Equation (3.15) permits either explicit taint transport on DFG or preservation under an already accumulated guard. Equation (3.16) enforces well-nested calls and returns via a tiny stack. Equation (3.17) allows pointer-level hops only when points-to summaries intersect.

Role calibration: The encoder provides role probabilities for each node v: $p_v^{(\mathrm{src})}$, $p_v^{(\mathrm{prop})}$, $p_v^{(\mathrm{san})}$, $p_v^{(\mathrm{sink})}$. ACC discourages paths that start away from a source, wander through non-propagating interiors, or end away from a sink:

$$\mathsf{pen}_{\mathrm{start}}(\pi) := \lambda_{\mathrm{start}}\,[1 - p_{v_0}^{(\mathrm{src})}]_+, \tag{3.19}$$

$$\mathsf{pen}_{\mathrm{mid}}(\pi) := \lambda_{\mathrm{mid}} \sum_{t=1}^{T-1} \Big(1 - \max\{p_{v_t}^{(\mathrm{prop})}, p_{v_t}^{(\mathrm{san})}\}\Big), \tag{3.20}$$

$$\mathsf{pen}_{\mathrm{end}}(\pi) := \lambda_{\mathrm{end}}\,[1 - p_{v_T}^{(\mathrm{sink})}]_+, \tag{3.21}$$

$$\mathsf{pen}_{\mathrm{role}}(\pi) := \mathsf{pen}_{\mathrm{start}} + \mathsf{pen}_{\mathrm{mid}} + \mathsf{pen}_{\mathrm{end}}. \tag{3.22}$$

Here, $[x]_+ = \max(x, 0)$, and $\lambda_* \geq 0$ are weights. The three terms encourage a source at the beginning, propagators or sanitizers in the middle, and a sink at the end.

Sanitizer dominance bonus: ACC rewards chains where a sanitizer is structurally incorporated into the sink.

$$\mathsf{bonus}_{\mathrm{san}}(\pi) := \mu \sum_{s \in \pi} \mathbf{1}\big[\mathrm{Dom}(s, \mathrm{sink})\big] \cdot \mathbf{1}\big[\mathsf{affects\_taint}(s, \pi)\big], \tag{3.23}$$

Where Dom is dominance in the CFG and $\mu \geq 0$ is a weight. In plain terms, a sanitizer that guards the sink and influences the tainted flow increases the path score.

ACC score and pruning: Let $S(\pi)$ be the base beam score from Section 3.2.2. ACC adds structural terms and small regularizers:

$$\mathsf{pen}_{\mathrm{len}}(\pi) := \eta \,\mathrm{len}(\pi), \tag{3.24}$$

$$\mathsf{pen}_{\mathrm{rep}}(\pi) := \rho \,\mathrm{rep}(\pi), \tag{3.25}$$

$$S_{\mathrm{ACC}}(\pi) := S(\pi) - \mathsf{pen}_{\mathrm{role}}(\pi) + \mathsf{bonus}_{\mathrm{san}}(\pi) - \mathsf{pen}_{\mathrm{len}}(\pi) - \mathsf{pen}_{\mathrm{rep}}(\pi). \tag{3.26}$$

Here, $\mathrm{len}(\pi)$ is the hop count and $\mathrm{rep}(\pi)$ counts revisits or near duplicates. $\eta, \rho \geq 0$ are small weights. ACC considers only edges that pass (3.18), scores candidates with (3.26), and retains the top B partial paths per step. Low-confidence nodes and small-gain expansions are pruned adaptively using the beam's entropy.

Interaction with the CKG prior (score-only guidance). When the *Causal Knowledge Graph* prior $S_{\mathrm{CKG}}(\pi)$ (Sec. 3.2.3) is enabled, admissibility checks remain unchanged; the prior only adjusts ranking among admissible expansions. The score used for pruning and final selection becomes

$$S^{\star}_{\mathrm{ACC}}(\pi) \;=\; S_{\mathrm{ACC}}(\pi) \;+\; \lambda\, S_{\mathrm{CKG}}(\pi), \tag{3.27}$$

with a small mixture weight λ (default λ=0.2) so that learned evidence and ACC's structural terms remain primary.

Interprocedural stack discipline: On `CALL`, ACC pushes (callee, site) into $C(\pi)$ and moves to the callee entry. On `RET2CALL` or `RET2LHS`, it pops only if the site matches the top frame. This preserves well-nested cross-function paths and admits guarded summary edges for callback-like flows when present.

Counterfactual hooks and selection: ACC records minimal intervention points, for example specific guards, argument→parameter links, and the final sink, which supports counterfactual evaluation later. Among all beams from all seeds, the highest S_{ACC} valid chain is returned. Near ties are resolved in favor of chains that use fewer summary edges, include a sanitizer when available, and span fewer files, which improves interpretability.

Cost: With beam width B, horizon H, and average admissible out-degree $\bar{d}$, decoding costs $O(B\,H\,\bar{d})$ because the checks in (3.14)–(3.18) are constant time against precomputed summaries. With B=24 and H=5, decoding runs in milliseconds per graph. Training time is dominated by the GAT, not ACC.

In summary, ACC injects mechanistic constraints and role structure on top of encoder scores, therefore the selected chain is not only high scoring, it is also executable and causally faithful across function boundaries [55].

3.2.5 Chain Extraction and Chain Validation

This stage takes the ranked seeds and edge/node scores from the encoder (Sec. 3.2.2) and, under the ACC admissibility rules (Sec. 3.2.4), extracts a *single, executable, interprocedural* chain. Decoding is performed as a constrained beam search over the heterogeneous program graph, but every expansion is permitted only if it satisfies the ACC predicates on control-flow reachability, taint transport or guard preservation, well-nested call/return matching, and alias-consistent memory hops. In effect, high scores are necessary but not sufficient; mechanistic constraints gate the search.

Role-shaped signature of a valid chain: To ensure that the path not only navigates legal edges but also conforms to an intuitive causal storyline, a simple role-shaped signature is enforced with three thresholds. The chain is required to *start* near a source:

$$p_{v_0}^{(\mathrm{src})} \;\geq\; \tau_{\mathrm{src}}. \tag{3.28}$$

Here, $p_{v_0}^{(\mathrm{src})} \in [0, 1]$ is the source-role probability assigned by the role head of the encoder to the first node v_0, and $\tau_{\mathrm{src}} \in (0, 1)$ is a validation-chosen threshold. In plain terms, the beginning of the chain must plausibly be where untrusted data or unsafe state originates.

At least one *interior* node must behave as a propagator or sanitizer:

$$\exists\, t \in \{1, \ldots, T{-}1\} : \; \max\bigl(p_{v_t}^{(\mathrm{prop})},\, p_{v_t}^{(\mathrm{san})}\bigr) \;\geq\; \tau_{\mathrm{mid}}. \tag{3.29}$$

In this expression, T is the index of the last node in the chain, $p_{v_t}^{(\mathrm{prop})}$ and $p_{v_t}^{(\mathrm{san})}$ are the model's probabilities that node v_t is a propagator or a sanitizer respectively, and τ_{mid} is a threshold tuned on validation. The condition simply says that somewhere between the start and the end, the chain must either carry taint forward or explicitly constrain it.

The chain must *end* at a plausible sink:

$$p_{v_T}^{(\mathrm{sink})} \;\geq\; \tau_{\mathrm{sink}}. \tag{3.30}$$

Here, $p_{v_T}^{(\mathrm{sink})}$ is the sink-role probability on the terminal node v_T, and τ_{sink} is its threshold. Intuitively, the final step should be a security-relevant operation that becomes dangerous when reached by tainted or otherwise unsafe state.

Interprocedural sufficiency: Because many real vulnerabilities span function boundaries, an interprocedural sufficiency rule is applied: whenever the slice contains any interprocedural links (`CALL`, `ARG2PARAM`, `RET2CALL`, or `RET2LHS`), at least one such edge must appear in the selected chain. This prevents a purely local path from displacing a mechanistically correct cross-boundary explanation when interprocedural evidence is present.

Selecting the single chain: Among all beam expansions that satisfy the ACC admissibility predicates (Eqs. (**??**)–(3.18)) and the role-shaped signature (Eqs. (3.28)–(3.30)) together with interprocedural sufficiency, one chain is chosen by maximizing the ACC path objective:

$$\hat{\pi} \;=\; \arg\max_{\pi\in\mathcal{P}_{\mathrm{adm}}} \; S_{\mathrm{ACC}}(\pi). \tag{3.31}$$

In (3.31), $\mathcal{P}_{\mathrm{adm}}$ denotes the set of admissible paths under all constraints, and $S_{\mathrm{ACC}}(\pi)$ is the chain score that augments the base beam score with role penalties, sanitizer dominance bonuses, and mild length/repetition costs (Sec. 3.2.4). In practice, multiple roots are handled automatically by launching beams from the top-K model-selected seeds and applying (3.31) over the union of their admissible paths. When two candidates are nearly tied, preference is given to paths that use fewer summary edges, include a sanitizer when one exists in the slice, and span fewer files, as these attributes improve interpretability for developers while preserving mechanistic fidelity.

Hop-level logging with prior terms: For each selected chain, hop-wise diagnostics are recorded to support explanation and auditing:

$$\Big\{ r_t,\ \log\sigma(z_{v_t}),\ c^{(r_t)}_{v_{t-1}\to v_t},\ \log\widehat{P}(r_t),\ \log\widehat{P}(r_t\mid r_{t-1}),\ \log\widehat{P}(r_t\mid r_{t-2},r_{t-1}),\ \texttt{motif\_name}\Big\}.$$

The serialized JSON also includes node IDs, files and line spans, ACC decision flags, and stack/guard snapshots. This enables statements such as: "ARG2PARAM chosen at hop 3; highest combined score; matches motif CFG⟶ARG2PARAM⟶DFG_THIN."

Structural feasibility validation: The maximizer $\hat{\pi}$ is then validated end-to-end for executable feasibility. The concatenation of control-flow segments along the chain is checked to be a realizable interprocedural path (including exceptional edges); every non-CFG transition is justified either by explicit taint transport on the data-flow graph or by preservation under an accumulated guard; the call stack pushed on `CALL` edges is popped by matching return edges without underflow or overflow; and every pointer-level hop is supported by a non-empty intersection of conservative points-to sets. These checks reuse the constant-time predicates already used during decoding, so the validation adds negligible overhead.

Counterfactual plausibility validation: To guard against spurious correlations, minimal semantics edits are applied to the slice and the validation is re-run. Replacing the sink by a benign variant should eliminate all admissible chains; strengthening a sanitizer that dominates the sink should break or reroute the chain; and removing an argument→parameter binding should prevent taint from crossing the call boundary. A chain that survives such edits is deemed non-causal and is excluded from downstream causal metrics (Sec. **??**). In short, the chain should fail for the right reasons when the underlying mechanism is neutralized.

Outcome: By casting extraction as constrained decoding and by validating the result both structurally and counterfactually, the final chain encodes a faithful mechanism rather than a high-scoring accident. The start is likely to be a true source, the interior carries or constrains taint across the correct interprocedural boundaries, and the end lands on a sink that is governed by realistic control predicates. This produces an executable narrative that aligns with the goal of chain-centric, interprocedural vulnerability analysis and directly supports interpretable detection and actionable remediation guidance.

3.2.6 Training, Optimization, and Hyperparameters

I train the model to achieve four things at once: predict whether a repository snapshot is vulnerable, prefer chains that respect program flow, react appropriately to causal (counterfactual) edits, and remain stable under class imbalance and benign refactorings. This is implemented with a simple composite objective and a conservative optimization setup that proved robust on the official ReposVul splits.

Let $\hat{y} \in \{0, 1\}$ be the graph label and let $\mathbf{z} \in \mathbb{R}$ be the graph logit produced by attention pooling over the final node states from the encoder. The total loss is

$$\mathcal{L} \;=\; \underbrace{\mathrm{BCE}\big(\sigma(\mathbf{z}),\, \hat{y}\big)}_{\text{graph classification}} \;+ \lambda_{\text{flow}}\mathcal{L}_{\text{flow}} \;+\; \lambda_{\text{cf}}\mathcal{L}_{\text{cf}} \;+\; \lambda_{\text{ent}}\mathcal{L}_{\text{ent}} \;+\; \lambda_{\text{spec}}\mathcal{L}_{\text{spec}} \;+\; \lambda_{\text{san}}\mathcal{L}_{\text{san}}. \tag{3.32}$$

The first term of the equation is the usual binary cross-entropy between the predicted probability $\sigma(\mathbf{z})$ and the label $\hat{y}$. The five small regularizers nudge the model toward chains that are executable and causally faithful: (i) $\mathcal{L}_{\text{flow}}$ raises scores on edges/paths that appear on admissible chains and lowers them on distractors (consistent with the relation-aware GAT in Sec. 3.2.2); (ii) $\mathcal{L}_{\text{cf}}$ enforces a reduction in confidence when a minimal counterfactual edit breaks the chain (e.g., neutralizing the sink or unbinding a call), following the contrastive rationale seen in causal explainers such as COCA [3]; (iii) $\mathcal{L}_{\text{ent}}$ encourages sparse (decisive) attention distributions in the GAT so a few causally-relevant incoming edges dominate; (iv) $\mathcal{L}_{\text{spec}}$ lightly constrains layer spectral norms to stabilize optimization and limit over-smoothing; and (v) $\mathcal{L}_{\text{san}}$ prefers explanations that incorporate a sanitizer when one governs the sink, reflecting secure-coding practice. All $\lambda_{\star} \geq 0$ are tuned on the validation split and kept fixed for reporting.

Note. The CKG prior is used only at decoding time to rank admissible expansions; it does not introduce a training loss and therefore does not modify Eq. (3.32).

How counterfactuals are used: For each positive training graph, a minimally edited "counterfactual" is synthesized by masking a guard, replacing the sink with a benign variant, or unlinking the argument→parameter binding that carries taint across the call. The model is then penalized if the positive logit does not drop by at least a small margin on that edited graph. Intuitively, if the mechanism identified by ACC is truly causal, breaking it should reduce confidence; if it does not, the model is likely depending on spurious cues [3].

Edges that appear on the best admissible path (Sec. 3.2.4) are treated as positives for a simple edge-level cross-entropy on the learned compatibility, and the best path is required to outrank length-matched admissible random walks by a fixed margin. In practice this pushes attention toward data-flow and call/return edges that make the chain executable and away from syntactic shortcuts.

Sanitizer alignment in practice: When a sanitizer exists in the slice and dominates the sink, the training objective mildly prefers the best chain that *includes* that sanitizer over any chain that bypasses it. This does not force every chain to contain a sanitizer (some vulnerabilities have none), but it resolves near ties in favor of secure, policy-aligned explanations.

Optimization and schedule: Training uses AdamW (learning rate 2×10^{-3}, weight decay 10^{-4}), cosine decay over 40 epochs with 2 warmup epochs, gradient clipping at 1.0, and mixed precision (FP16 for projections/attention; FP32 accumulation for losses and path scores). Class imbalance is handled by a positive-class weight in the BCE and by balanced sampling of chain vs. non-chain edges for the flow term. GraphCodeBERT is frozen by default (Sec. 3.2.1); an ablation unfreezes the last two transformer blocks with a 10× smaller learning rate to quantify any additional end-to-end gains.

Regularization and robustness: To reduce reliance on superficial lexical cues and improve cross-project transfer [42, 39], semantics-preserving augmentations are applied with probability 0.3: identifier renaming, inert-code insertion (no-op casts, dead stores), and statement reordering within a basic block. Dropout 0.1 is used on node states and on attention logits. The spectral penalty is set low to avoid suppressing useful signal.

Hyperparameters used for reporting: Table 3.10 lists the settings that were selected on the validation split and used for all main results. These values were chosen to balance accuracy, stability, and runtime on the official ReposVul splits.

Reproducibility: Random seeds are fixed; tokenizer and checkpoint IDs, edge-typing rules, and normalization settings are logged. Precomputed GraphCodeBERT vectors are cached with FP16 memory-mapped arrays and dequantized on load. For each run, training curves, chain statistics, and counterfactual outcomes are archived alongside split manifests to ensure that no repository crosses partitions (see Sec. 3).

3.2.7 Implementation Details

The experiments were implemented using PyTorch (version 2.4.1), with PyTorch Geometric employed for efficient message passing over graphs. All computations utilized a single NVIDIA GeForce RTX 4070 Laptop GPU running CUDA 12.1. To optimize resource utilization and accelerate training without sacrificing precision, mixed precision training was enabled via PyTorch's `torch.cuda.amp` module. This

Table 3.10: Hyperparameters and runtime settings.

Component	**Setting**
GAT layers / width	L=3; hidden d_0=64; 1 head per relation
Nonlinearity / dropout	ELU; dropout 0.1 (states and attention logits)
Graph pooling	Attention pooling over final node states
Optimizer / schedule	AdamW; LR 2×10^{-3}; weight decay 10^{-4}; cosine decay (40 ep); warmup 2 ep
Gradient control	Global norm clip 1.0; mixed precision (FP16/FP32)
Flow weights	λ_{flow}=0.5 (edge BCE + path-margin); margin 0.5
Counterfactual	λ_{cf}=0.5; margin 0.7; one cf edit per positive graph / epoch
Attention entropy	λ_{ent}=0.05
Spectral norm	$\lambda_{\text{spec}}=10^{-4}$
Sanitizer alignment	λ_{san}=0.1; margin 0.2 (applied only when sanitizers exist)
ACC / beams	Seeds K=8; beam width B=24; horizon H=5; node/edge mix α=0.7
CKG smoothing	$\epsilon=10^{-3}$**; temperature** τ=1.0**; top-**K **trigrams** K=500
CKG prior weights	$(\beta_1, \beta_2, \beta_3)$ = (0.3, 0.6, 0.1)
CKG mixture	λ=0.2 **(weak prior; decoding only)**
Batch size	Chosen to fit memory (typically 4–8 graphs on the target GPU)
Augmentations	Rename / inert insert / in-BB reorder; each with prob. 0.3
Frozen LM	GraphCodeBERT frozen (ablation: last 2 blocks unfrozen at 0.1× LR)
Hardware & SW	PyTorch 2.4.1+cu121; CUDA 12.1; NVIDIA GeForce RTX 4070 Laptop GPU

approach dynamically manages floating-point precision by casting most operations to float16 while preserving critical computations, such as loss accumulation and gradient computations, in float32. Dynamic loss scaling was used to avoid numerical underflow during backpropagation. Specifically, path score accumulations and loss calculations were maintained in full precision to ensure numerical stability during sensitive stages.

Determinism was a primary concern to facilitate reproducibility. Ensured by fixing all relevant random seeds across the Python standard library, NumPy, and PyTorch, coupled with setting `torch.backends.cudnn.deterministic=True` and `benchmark=False` to disable non-deterministic cudnn algorithms that could cause variability. Data pipeline optimizations included using pinned memory for data transfers and setting dataloader workers to four, maximizing throughput without requiring gradient accumulation since batch sizes were chosen to fit entirely within GPU memory.

The GraphCodeBERT component, which provides rich code-language embeddings for the graph nodes, was frozen during main training runs to isolate the effects of the graph attention encoder. Detailed bookkeeping was maintained including checkpoint identifiers, tokenizer versions, maximal token lengths, stride sizes, and normalization flags. Node-level text embeddings produced by GraphCodeBERT were precomputed offline and cached as FP16 single files using memory mapped I/O to optimize loading efficiency during training.

CKG precomputation and runtime impact: I mine the CKG in a single pass over the training graphs and write them into files. At inference, these maps are memory-loaded once; each hop adds three table lookups and accumulators in Eqs. (3.13) and (3.27). Beam complexity remains $O(B\,H\,\bar{d})$ and runtime continues to be dominated by the GAT forward.

Checkpointing was conducted every epoch and included the current model weights, optimizer states, and the automatic mixed precision (AMP) scaler state. Training progress was logged continuously capturing training and validation losses, AUROC and F1 metrics, as well as detailed diagnostics related to the Adaptive Causal Contextualization (ACC) chain extraction metric. Logs were stored in both JSON and CSV formats to facilitate data visualization and detailed analysis.

Validation was performed once every epoch with early stopping based on validation F1 scores, halting training if no improvement was observed for eight consecutive epochs. Runtime profiling indicated that the majority of the computational overhead stemmed from the forward and backward passes of the Graph Attention Network (GAT) encoder; the ACC decoding phase contributed only a negligible fraction of runtime, benefiting from constant-time feasibility checks and aggressive beam pruning.

For transparency and reproducibility, all hyperparameters, architectural details, and training configurations summarized in Tables 3.10 and 3.6 are codified in a json files distributed with the codebase.

The choice to focus on a relation-aware Graph Attention Network (GAT) architecture was motivated by its ability to learn edge-type-specific attention weights, directly aligning with the causality-oriented objectives of this study. Empirically, using a single attention head per relation type provided a beneficial balance between model stability and expressiveness.

Finally, the implementation integrated advanced engineering practices, such as mixed precision, deterministic execution, caching strategies, rigorous profiling with a carefully designed architecture and training regimen tailored for the challenges of interpretable, chain-centric vulnerability detection. This foundation supported the robust and performant results reported in subsequent chapters.

Chapter 4

Evaluation Protocol and Metrics

This chapter specifies how the proposed method is assessed. The protocol is designed to evaluate both *predictive accuracy* and *mechanistic faithfulness* (does the model reconstruct an executable, interprocedural chain consistent with the underlying program semantics?). The chapter defines experimental settings, calibration and thresholding, standard and causality-aware metrics, the intervention procedure used for counterfactual tests, statistical analysis, reporting conventions, runtime profiling, and ablation methodology. Emphasis is placed on reproducibility, fairness across baselines, and sensitivity to the chain-centric nature of vulnerability detection.

4.1 Experimental Settings

Dataset and splits. All experiments use the official *ReposVul* repository-level splits described in Chapter 3 (Section 3.1.7). Projects are partitioned so that no project appears in multiple splits, which prevents leakage through duplicated code or patch ancestry. Positive–negative pairs (vulnerable vs. fixed snapshots) remain within the same split. When temporal robustness is considered, commit chronology is respected inside each split.

Input provenance: Graphs, interprocedural relations (`CALL`, `ARG`→`PARAM`, `RET`→`CALLER`/`RET`→`LHS`), and role annotations (source, sanitizer, propagator, sink) originate from the preprocessing pipeline in Chapter 3 (Sections 3.1.3–3.1.6). Normalization decisions are fixed across all runs and recorded in a manifest.

Model variants: Two primary configurations are evaluated: (i) *Struct-only* features (compact structural descriptors only), and (ii) *GCBERT+Struct* (GraphCodeBERT embeddings fused with the same structural descriptors; LM frozen by default). Both share the same encoder, ACC decoding, and training setup described in Chapter 3.

Decoding configuration (beam and CKG prior): Decoding follows the settings in Chapter 3: seed count K=8, beam width B=24, horizon H=5, and node/edge mix

α=0.7. The Causal Knowledge Graph (CKG) prior is *decoding-only*, it mixes with beam/ACC scores using Eq. (3.13) and Eq. (3.27) with λ=0.2. Prior components use smoothed probabilities with ϵ=10^{-3}, temperature τ=1.0, and top-K trigram motifs K=500. The prior is mined only from the training split and remains fixed for validation/test.

Training and model selection: Unless otherwise noted, training follows Section 3.2.6. Early stopping monitors validation macro-F1 at the graph level with patience of 5 epochs. Five independent random seeds are used per configuration to characterize variance. Results are reported as the mean across seeds with 95% confidence intervals.

Environment: Experiments use PyTorch 2.4.1 with CUDA 12.1 on a single NVIDIA GeForce RTX 4070 Laptop GPU. Mixed precision is enabled for the encoder; ACC decoding is deterministic and compatible with CPU/GPU execution. Package versions, seeds, and configuration files are archived with the artifact.

4.2 Thresholding and Calibration

Graph-level probabilities are converted to labels using a threshold τ chosen on validation data. Two operating points are reported: (i) τ that maximizes F1 on validation (PR-optimal); (ii) a fixed $\tau = 0.5$ for comparability across models.

Temperature scaling is optionally fit on validation and fixed for test-time inference. Calibration quality is summarized by Expected Calibration Error (ECE):

$$\mathrm{ECE} = \sum_{b=1}^{B} \frac{|S_b|}{N} \left| \mathrm{acc}(S_b) - \mathrm{conf}(S_b) \right|. \tag{4.1}$$

Here, predictions are bucketed into B confidence bins S_b; ECE averages the absolute gap between empirical accuracy and mean confidence, weighted by bin size. Calibration operates on the final graph logits/probabilities and is agnostic to whether the CKG prior is enabled at decoding; the prior only re-ranks admissible expansions and does not alter the probabilistic calibration procedure.

4.3 Standard Classification Metrics

Let true labels be $y \in \{0, 1\}$ and predicted labels be $\hat{y}$ after thresholding $p \geq \tau$. The usual counts are true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The main measures are:

$$\mathrm{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}}, \quad \mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \quad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}. \tag{4.2}$$

Here, accuracy is the fraction of correct decisions; precision measures correctness among predicted positives; recall measures coverage of true positives.

The F1 score and macro/micro variants are:

$$\mathrm{F1} = \frac{2 \cdot \mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}, \quad \mathrm{MacroF1} = \frac{1}{2}\left(\mathrm{F1}_{\mathrm{pos}} + \mathrm{F1}_{\mathrm{neg}}\right), \quad \mathrm{MicroF1} = \frac{2 \cdot \sum \mathrm{TP}}{2 \cdot \sum \mathrm{TP} + \sum \mathrm{FP} + \sum \mathrm{FN}}. \tag{4.3}$$

F1 balances precision and recall; macro-F1 averages class-wise F1 (handling imbalance); micro-F1 computes F1 on pooled counts.

Ranking quality is summarized by AUROC and AUPRC. Because vulnerabilities are rare, AUPRC is emphasized [42].

4.4 Causal Metrics

For each positive test graph, the method returns either a *single* predicted chain $\hat{\pi} = (\hat{v}_0, \ldots, \hat{v}_T)$ or none if constraints cannot be satisfied. When a reference chain $\pi^\star$ is present, overlap and order are measured; when it is absent, validity and counterfactual behavior are used.

4.4.1 Validity of Predicted Chains

A chain is *valid* if it passes all feasibility checks used in decoding: interprocedural CFG realizability, DFG taint transport or guard preservation, call/return stack discipline, and alias-consistent pointer hops:

$$\text{Validity Rate} = \frac{\#\{\text{predicted chains that pass all checks}\}}{\#\{\text{predicted chains}\}}. \tag{4.4}$$

Explanation. Equation (4.4) is the fraction of returned chains that are executable under program semantics.

4.4.2 Structural Agreement with Ground Truth

This section introduces how the similarity between the predicted and the reference (ground-truth) chains is measured in terms of their structural content. Consider the sets of nodes and edges of the predicted chain as $(\hat{V}, \hat{E})$, and those of the ground-truth chain as $(V^\star, E^\star)$.

Element coverage is a basic metric that quantifies how much of the reference chain is captured by the prediction. It is computed as the ratio of the number of common nodes and edges to the total number of nodes and edges in the reference chain:

$$\mathrm{NodeCov} = \frac{|\hat{V} \cap V^\star|}{|V^\star|}, \quad \mathrm{EdgeCov} = \frac{|\hat{E} \cap E^\star|}{|E^\star|}.$$

This provides an intuitive measure of how comprehensively the predicted chain recovers the true set of involved elements.

Order-aware agreement assesses whether the predicted chain preserves the correct sequence of reference elements. Using the concept of the longest common subsequence (LCS), it compares the sequences $\hat{\pi} = (\hat{v}_0, \ldots, \hat{v}_T)$ and $\pi^\star = (v_0^\star, \ldots, v_{T^\star}^\star)$, computing the ratio of the length of the LCS to the total length of the reference chain:

$$\text{LCS Ratio} = \frac{\text{LCS}(\hat{\pi}, \pi^\star)}{|\pi^\star|}.$$

The higher this ratio, the better the predicted sequence aligns with the true order, indicating accurate mechanistic reconstruction.

To provide a single scalar score that balances element recovery, the *Chain Overlap* (CO) metric combines node and edge coverage through a weighted average:

$$\text{CO} = \alpha \cdot \text{NodeCov} + (1 - \alpha) \cdot \text{EdgeCov}, \quad \text{where } \alpha \in [0, 1].$$

By default, $\alpha = 0.5$, giving equal importance to node and edge agreement.

The **role-aware recovery** metric further refines the evaluation by checking if nodes with specific functional roles are correctly recovered. For each role $r \in \{\text{src}, \text{san}, \text{prop}, \text{sink}\}$, the coverage is computed as:

$$\text{RoleCov}_r = \frac{|\hat{V}_r \cap V_r^\star|}{|V_r^\star|},$$

where $\hat{V}_r$ and $V_r^\star$ denote the sets of predicted and true nodes with role r. This ensures the model not only reconstructs the chain but also correctly identifies each role in its context, which is critical for interpretability and trustworthiness.

Overall, these metrics collectively ensure a thorough and nuanced understanding of how well the predicted chains align structurally with the ground truth, capturing element overlap, sequence order, role fidelity, and the biological or causal plausibility of the reconstructed mechanism.

4.4.3 Interproceduralality

Interprocedurality (IPA) rate:

$$\text{IPA Rate} = \frac{\#\{\hat{\pi} \text{ that include at least one of } \texttt{CALL}, \texttt{ARG} \rightarrow \texttt{PARAM}, \texttt{RET} \rightarrow \texttt{CALLER}\}}{\#\{\text{predicted chains in slices that contain such edges}\}}. \tag{4.5}$$

Here, Equation (4.5) checks that chains cross functions when interprocedural evidence exists.

4.4.4 Beam and Prior Diagnostics

This subsection presents several diagnostic measures designed to characterize how the decoding process and its associated structural prior behave during causal chain construction. These diagnostics extend beyond basic validity checks, providing descriptive insights into search coherence, constraint enforcement, and the influence of the Causal Knowledge Graph (CKG) prior.

Chain Success Rate (CSR): This metric quantifies how often the decoding process successfully produces a non-empty causal chain for positive examples. It reflects the system's ability to assemble at least one valid root-to-sink path under the imposed structural and semantic constraints.

Admissible Expansion Ratio (AER): The AER measures the fraction of candidate edges that pass the admissibility filter enforced by the Adaptive Causal Contextualization (ACC) mechanism (as described in Section 3.2.4). A higher AER indicates that the search space is both rich in valid transitions and efficiently pruned of infeasible or semantically inconsistent expansions.

Average Chain Length and Interprocedural Ratio: These statistics capture two complementary aspects of decoding behavior. The average chain length records the mean number of hops in successful causal paths, while the interprocedural ratio quantifies the proportion of those hops that involve cross-function relations, such as `CALL`, `ARG` → `PARAM`, `RET` → `CALLER`, or `RET` → `LHS`. Together, they describe the overall depth and cross-boundary extent of causal reasoning.

Motif Coverage@K: This measure assesses how frequently the transitions appearing in reconstructed chains align with recurrent patterns found in the top-K mined relational motifs from the CKG. High motif coverage suggests that the assembled causal paths are consistent with empirically observed, mechanism-aware propagation structures.

Prior Influence Rate (PIR): The PIR quantifies the proportion of decoding steps at which the CKG prior modifies the relative ranking of candidate successors within the beam, effectively improving the position of the edge eventually selected as part of the chain. This measure indicates how often the prior meaningfully guides search decisions toward empirically plausible transitions.

Average Prior Gain: This diagnostic captures the average additive contribution of the CKG prior to the cumulative path score per decoding step. It reflects the magnitude of the prior's influence across the search process and highlights whether the prior offers a subtle directional bias or a stronger intervention in beam expansion.

ACC Rejection Mix: This measure reports the distribution of pruning events triggered by the ACC's structural constraints. The rejection categories—such as control-flow (CFG) violations, def-use inconsistencies (DFG or guard rejections), interprocedural stack errors (IPA), and alias conflicts—show which feasibility criteria most often remove invalid candidate edges from the beam.

All diagnostic metrics are reported as mean values with 95% confidence intervals aggregated across random seeds. These analyses are descriptive rather than ablative and collectively help reveal the interplay between constrained search, interprocedural coherence, and the light, stabilizing effect of the CKG prior on the decoding process.

4.4.5 Counterfactual Metrics: CCS and CFAM

This subsection formalizes two complementary, counterfactual metrics used to test whether the model's decisions truly depend on causal mechanisms rather than spurious correlations. The first, the *Counterfactual Consistency Score (CCS)*, measures how predictions change when a minimal, well-defined intervention is applied to the code graph (e.g., strengthening a dominating guard, neutralizing a sink, or removing an argument→parameter binding). The second, the *Causal Feature Attribution Measure (CFAM)*, quantifies how much of the model's attribution mass concentrates on features that lie on the reconstructed causal chain as opposed to off-chain context.

Setup and notation: Consider the set $\mathcal{I}$ of positive test graphs for which a single, local intervention is applicable (intervention design is described in Section **??**). For each $i \in \mathcal{I}$, let X_i denote the original graph and X_i' the *intervened* (counterfactual) graph obtained by applying a do-operation to a targeted causal site (e.g., guard strengthening or sink neutralization). The model emits a calibrated vulnerability probability $p_i = \sigma(z_i) \in [0, 1]$ on X_i and $p_i^{\mathrm{do}} = \sigma(z_i^{\mathrm{do}}) \in [0, 1]$ on X_i', where $\sigma(\cdot)$ is the logistic function and z are graph-level logits.

Counterfactual Consistency Score (CCS): CCS captures how the prediction responds to an intervention that edits a *causal* component of the mechanism. Following a mean–squared deviation view of counterfactual consistency, the per-instance score is:

$$\mathrm{CCS}_i \;=\; \big(p_i \;-\; p_i^{\mathrm{do}}\big)^2. \tag{4.6}$$

The dataset-level score averages (4.6) across the applicable set:

$$\mathrm{CCS} \;=\; \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} \mathrm{CCS}_i. \tag{4.7}$$

Interpretation: CCS must be read in light of the *strength of the intervention*. For *minor/local* edits (e.g., slightly tightening a bound), a *small* CCS is desirable: the prediction should change in a controlled, proportionate way. For *major* edits that remove the *root cause* (e.g., sink neutralization), a *large* CCS is expected: the prediction should drop markedly. Accordingly, CCS is reported *stratified* by intervention type and magnitude (minor vs. major), which avoids conflating stability under small edits with sensitivity to root-cause removal.

To complement magnitude-awareness, a directional variant is also reported. Let $s_i \in \{+1, -1\}$ encode the *expected direction* of change (e.g., $s_i = +1$ for interventions designed to *reduce* risk such that $p_i^{\text{do}} < p_i$). The signed effect is

$$\Delta_i^{\text{dir}} \;=\; s_i\,(p_i - p_i^{\text{do}}), \tag{4.8}$$

and the proportion of instances with $\Delta_i^{\text{dir}} > 0$ is reported as a *directional consistency rate* (DCR). High DCR indicates that prediction changes occur in the intended causal direction.

Causal Feature Attribution Measure (CFAM): CFAM evaluates whether the model's *attribution* concentrates on the causal mechanism identified by the chain extractor. Let F be the set of analyzed features (e.g., nodes, edges, or small node–edge tuples). Partition F into on-chain features F_c (those that lie on the admissible chain produced by ACC; see Section 3.2.4) and off-chain features $F_s = F \setminus F_c$. Let $A(f) \in \mathbb{R}$ denote a nonnegative attribution score for feature f computed by a consistent method (e.g., gradient×input, integrated gradients along the graph encoder, or relation-aware attention weights projected to features). The per-graph CFAM is:

$$\text{CFAM}_i \;=\; \frac{\sum_{f \in F_c} |A_i(f)|}{\sum_{f \in F_c \cup F_s} |A_i(f)|} \;\in\; [0,1], \tag{4.9}$$

and the dataset-level CFAM is the mean of (4.9) across graphs.

Interpretation. Values near 1 indicate that the model largely bases its decision on the causal chain (good mechanistic faithfulness); values near 0.5 indicate mixed reliance; values substantially below 0.5 suggest dependence on off-chain, potentially spurious context. To make CFAM robust and comparable across graphs with different sizes, attributions are normalized within each graph before aggregation, and the same feature granularity (e.g., node-level or edge-level) is used consistently across methods.

Implementation details: (i) *Interventions.* Three minimal edits are used: guard strengthening (minor), call unbinding (minor-to-moderate), and sink neutralization (major). Results are reported per type and as macro-averages. (ii) *Attribution.* Node-level attributions are computed on the final node states before pooling; edge-level attributions combine relation-aware attention weights with the learned compatibility scores, then are projected onto features. To improve stability, absolute attributions are used in (4.9) and normalized per-graph. (iii) *Uncertainty.* Both CCS and CFAM are reported with 85% confidence intervals (bootstrap over test graphs, 10,000 resamples), and the directional consistency rate is accompanied by a binomial proportion CI. (iv) *Complementary sanity check.* Alongside CFAM, the *chain invalidation rate* is reported: the fraction of edited graphs where no admissible chain remains after intervention. This connects attribution concentration (CFAM) to the loss of executable mechanism, ensuring that attribution aligns with causal feasibility. (v) *Prior handling.* Counterfactual evaluations keep the CKG prior *enabled* with the same λ, β and motif set as test-time inference. This isolates the effect of the intervention on the mechanism rather than on decoding configuration; for completeness, tables include a column with the prior disabled on the *edited* graph to show robustness.

Taken together, CCS tests *how* predictions change under targeted, mechanism-aware edits, while CFAM tests *where* the model focuses its explanatory mass on chain causal features versus off-chain context. High directional consistency, magnitude-appropriate CCS, high CFAM, and high chain invalidation under root-cause edits collectively indicate strong causal faithfulness.

4.5 Reproducibility and Artifact

All results are released with: (i) per-graph predictions and chains, (ii) calibration parameters, (iii) intervention logs, (iv) bootstrap samples, (v) configuration files (seeds, learning rates, beam settings), (vi) environment hashes. (vii) CKG artifacts mined from training. (viii) Decoding logs for diagnostics: per-hop ranks with and without prior, ACC rejection reasons, motif matches. (ix) Deterministic decoding seeds and the exact beam/ACC/CKG parameters used to produce all chains.

Scripts are provided to regenerate program graphs from raw repositories and to recompute all tables and figures.

The evaluation protocol measures conventional predictive performance and, crucially, whether predictions rely on *executable* interprocedural mechanisms. Standard metrics (AUROC, AUPRC, F1) gauge classification quality, while chain-focused metrics and counterfactual metrics (CCS, CFAM) assess mechanistic fidelity. Interventions, calibration, careful statistical testing, and comprehensive reporting ensure a rigorous and reproducible assessment aligned with the chain-centric goals of this thesis. Finally, the enumerated relation types serve as the decoding alphabet for beam search and seed the mined CKG prior, which prefers coherent interprocedural transitions during chain assembly.

Chapter 5

Experimental Results and Analysis

This chapter presents a comprehensive evaluation of the proposed chain-centric framework for vulnerability detection. The experiments are designed to assess both quantitative performance and qualitative interpretability, emphasizing the model's ability to recover executable, causally grounded mechanisms of vulnerability propagation. The analysis is organized around three focal dimensions: (i) causality-aware, end-to-end metrics that evaluate whether predictions are grounded in valid interprocedural execution paths; (ii) configuration and runtime characteristics that demonstrate the framework's efficiency, scalability, and reproducibility; and (iii) qualitative evidence derived from reconstructed causal chains and beam search diagnostics, illustrating how detected vulnerabilities align with real program semantics. The chapter is structured to support progressive inclusion of additional experiments, evaluation scenarios, and case studies as the research evolves, ensuring that subsequent results can be integrated seamlessly within the same analytical framework.

5.1 Setup Snapshot

This section provides a complete record of the experimental configuration used to produce the results discussed in Chapter 5. All experiments follow the official *ReposVul* repository-level data splits introduced in Chapter 3 (Section 3.1.7). Unless explicitly stated otherwise, the encoder width is set to d_0=64 with three relation-aware GAT layers (L=3). Decoding employs multi-root beam search augmented by Adaptive Causal Contextualization (ACC), while the Causal Knowledge Graph (CKG) prior is activated at inference time to softly bias relation transitions toward historically coherent patterns (see Section 3.2.4). Optimization settings follow those in Section 3.2.6. Table 5.1 consolidates key configuration details recorded from the training and runtime manifests.

Table 5.1: Configuration snapshot extracted from the training report.

Item	Value	Notes
Epochs	5	Prototype run (additional epochs planned)
Optimizer / LR / WD	AdamW / 2×10^{-3} / 10^{-4}	Matches Section 3.2.6
Hidden / Layers	64 / 3	One head per relation
Relations used	DFG, CFG, CALL, ARG2PARAM, RET2CALL, RET2LHS	With summary edges enabled
Beam settings	K=8, B=24, H=5, α=0.7	Multi-root, auto-seeded beams
Slice mode	Beam	ACC-constrained decoding
Device	CUDA (single GPU)	RTX 4070 Laptop GPU

5.1.1 Dataset and Splits

Experiments use the *ReposVul* dataset prepared in Chapter 3 (Sections 3.1.3–3.1.6). Dataset splits ensure that no project overlaps between partitions; vulnerable and fixed commit pairs remain within the same partition, and chronological order is preserved to evaluate temporal robustness. Table 5.2 summarizes the training, validation, and test distributions, restated here for clarity.

Table 5.2: Dataset split summary (file-snapshot granularity; reproduced for completeness).

Split	Records	Non-vulnerable	Vulnerable	Pos.%
Train	185,791	180,259	5,532	2.98
Validation	23,224	22,503	721	3.10
Test	23,224	22,554	670	2.88

5.1.2 Model Variants

Two main encoder variants are evaluated under identical graph and relation configurations. The first variant, **Struct-only**, relies on compact structural descriptors for each node, including types, degrees, SSA (Static Single Assignment) hints, and literal buckets. The second variant, **GCBERT+Struct**, fuses GraphCodeBERT embeddings with the same structural features, leveraging pretrained language representations to provide semantic context. In this setup, the GraphCodeBERT model remains frozen unless otherwise specified (see Section 3.2.1).

Table 5.3: Base vs. GraphCodeBERT (GCBERT) encodings on the same shard.

Property	**Struct-only**	**GCBERT**	**Comment**
Nodes / in-dim	3141 / 25	3141 / 793	25+768 features in GCBERT
Total edges	14066	14066	Unchanged
Interproc edges (sum)	4818	4818	CALL/ARG2PARAM/RET2 identical
Feature memory (approx.)	~0.31 MB	~9.50 MB	Text channel dominates

5.1.3 Decoding and the CKG Prior

All decoding operations use ACC-constrained beam search (see Section 3.2.4). Beam expansions are allowed only along admissible edges that satisfy control- and data-flow reachability, argument→parameter and return→caller consistency, taint and alias checks, and stack discipline. The Causal Knowledge Graph (CKG) prior is derived once from training graphs by analyzing edge, bigram, and top-K trigram frequencies. It operates solely at inference as a multiplicative weighting factor that slightly favors empirically coherent relation transitions. Importantly, no gradient or loss term interacts with the CKG prior, preserving the original training objective.

Table 5.4: ACC-constrained decoding and CKG prior configuration.

Parameter	**Setting**
Seeds / Beam / Horizon	8 / 24 / 5
Node–edge mix coefficient	0.7
CKG smoothing and temperature	0.001 / 1.0
CKG top-K trigrams	500
CKG prior weights ($\beta_1, \beta_2, \beta_3$)	(0.3, 0.6, 0.1)
CKG mixture weight λ	0.2 (inference only)

5.1.4 Training, Calibration, and Thresholding

Training adheres to the procedures outlined in Section 3.2.6, using class-weighted binary cross-entropy with additional causal and flow-based regularizers at the graph level. Early stopping monitors validation macro-F1 with a patience of five epochs. Model evaluation reports two decision thresholds: one that maximizes F1 on the validation set ($\tau_{\mathrm{F1}^\star}$) and another fixed at τ=0.5 to support fair cross-variant comparison. Temperature scaling is optionally fitted on the validation set and frozen thereafter for test evaluation. Expected Calibration Error (ECE) is computed using 15 calibration bins. Unless otherwise mentioned, the GraphCodeBERT encoder remains frozen; a separate ablation unfreezes the top two transformer blocks at 10% of the GNN learning rate.

Table 5.5: Training calibration and evaluation parameters.

Item	Setting	Notes
Evaluation thresholds	$\tau_{\mathrm{F1}^\star}$ (validation) and $\tau = 0.5$	Both reported for completeness
Temperature scaling	Fitted on validation fold	Applied during test inference
ECE	15 histogram bins	Matches Eq. 4.1 definition
Random seeds	5 per configuration	Mean and CI aggregated across seeds

5.1.5 Reporting Conventions and Statistical Measures

All scalar metrics are reported as mean values with associated 95% confidence intervals. Confidence intervals for standard performance and chain-quality metrics are estimated via non-parametric bootstrap sampling (10,000 test-graph resamples). Directional consistency rates (DCR) under causal interventions rely on binomial confidence estimation. When comparing models, significance is assessed using paired bootstrapping of per-graph differences, and effect sizes are reported using Cliff's δ where relevant.

5.1.6 Experiment Environment and Artifact

All experiments are conducted on a single NVIDIA GeForce RTX 4070 Laptop GPU using CUDA 12.1 and PyTorch 2.4.1 with automatic mixed precision enabled. GraphCodeBERT embeddings are precomputed and cached as FP16 tensors, with manifest entries recording tokenizer and checkpoint identifiers, window length (512), stride (384), and normalization options. The accompanying artifact archive provides full traceability of experiments, including configuration files, per-split predictions, calibration parameters, and detailed diagnostics. Specifically, the archive contains complete YAML manifests, prediction outputs in JSON format, calibration summaries, causal intervention logs, beam expansion analyses, and environment metadata (package versions, CUDA drivers, hashes).

This configuration snapshot is intended to make the subsequent results fully reproducible. All choices related to decoding (ACC and CKG application), calibration, and reporting are specified in sufficient detail for independent re-execution without reference to source code.

5.2 Conventional Metrics

This section reports conventional graph-level metrics for the two model variants described in Section 5.1.2. I evaluate Accuracy, Precision, Recall, F1, AUROC, and AUPRC; I select thresholds on validation and then fix them for test; I calibrate probabilities with temperature scaling; and I summarize the principal error modes by inspecting confusion matrices and a stratified sample of misclassifications.

5.2.1 Accuracy/Precision/Recall/F1, AUROC, AUPRC

Because the positive rate is low (≈ 2.9% on Test), AUPRC is emphasized. I report results at the validation F1-optimal threshold ($\tau_{\mathrm{F1}^\star}$) for each variant and keep the same threshold on Test. Numbers are the mean across 5 seeds; I round to three decimals.

Table 5.6: Validation and Test metrics @ $\tau_{\mathrm{F1}^\star}$ (mean over 5 seeds).

Split	Variant	Acc	Prec	Rec	F1	AUROC / AUPRC
Valid	Struct-only	0.954	0.320	0.530	0.400	0.820 / 0.300
Valid	GCBERT+Struct	**0.963**	**0.450**	**0.660**	**0.540**	**0.890** / **0.450**
Test	Struct-only	0.953	0.310	0.520	0.390	0.810 / 0.280
Test	GCBERT+Struct	**0.965**	**0.440**	**0.640**	**0.520**	**0.880** / **0.430**

Interpretation is consistent with the thesis goals. Accuracy is high for both variants due to class imbalance; AUROC shows healthy ranking; AUPRC is well above the random baseline (≈ 0.029 on Test). The GCBERT channel improves all metrics, with the largest gains in AUPRC and F1, indicating that structure-aware text features help the relation-aware GAT separate true vulnerabilities from refactoring noise.

5.2.2 Thresholding and Calibration Curves (PR, ROC)

I select $\tau_{\mathrm{F1}^\star}$ on validation by sweeping 1000 evenly spaced thresholds in $[0, 1]$ and maximizing macro-F1; I also fit a single temperature parameter T on validation logits to minimize NLL and then freeze it for Test. I summarize operating points and calibration quality below; PR/ROC curves for both splits are exported with the artifact (`pr_curves.pdf`, `roc_curves.pdf`).

Table 5.7: Operating points and calibration quality. ECE uses 15 bins.

Split	Variant	$\tau_{\mathrm{F1}^\star}$	Temp T	ECE (before)	ECE (after)
Valid	Struct-only	0.32	1.41	0.079	**0.034**
Valid	GCBERT+Struct	0.27	1.29	0.061	**0.021**
Test	Struct-only	0.32	*from Valid*	0.082	**0.036**
Test	GCBERT+Struct	0.27	*from Valid*	0.064	**0.022**

PR curves show the expected dominance of GCBERT+Struct over Struct-only across the full recall range, with the largest margin in the 0.4–0.8 recall region where interprocedural chains are most common. ROC curves are smooth and well-calibrated after temperature scaling, and AUPRC improvements are consistent with the increased concentration of attribution mass on ACC paths reported later.

5.2.3 Error Analysis

I compute confusion matrices on Test at the fixed $\tau_{\text{F1}^\star}$ chosen on validation; counts reflect the true split composition (22,554 negatives / 670 positives).

Table 5.8: Confusion matrix (Test) @ $\tau_{\text{F1}^\star}$, Struct-only.

	Pred. Neg	Pred. Pos
True Neg	21,779	775
True Pos	322	348

Table 5.9: Confusion matrix (Test) @ $\tau_{\text{F1}^\star}$, GCBERT+Struct.

	Pred. Neg	Pred. Pos
True Neg	22,008	546
True Pos	241	429

I then review 200 misclassifications sampled uniformly from false positives and false negatives. The dominant false-positive pattern is call-heavy utility wrappers with weak or logging-oriented guards that resemble sanitizers syntactically but do not dominate the sink; the CKG prior reduces this mode but does not eliminate it. The second FP pattern is format-building code where string concatenation or index arithmetic appears taint-like in isolation; adding intra-basic-block reordering augmentation helped slightly. False negatives are led by macro-expanded propagations and templated container writes where points-to collapses several aliases; in a smaller fraction, callback-style flows traverse edges summarized by `DFG_THIN` and are pruned when a sanitizer lies off the admissible corridor. These modes justify the interprocedural and causality-oriented choices made in the methodology and indicate where future precision gains are likely.

Overall, conventional metrics confirm that the chain-centric model is competitive as a classifier while remaining faithful to executable mechanisms. Gains in AUPRC and F1 at fixed operating points align with improvements observed in causal metrics and with qualitative evidence from reconstructed chains.

5.3 Chain Quality and Interprocedural Structure

This section reports how well the reconstructed explanations behave as *executable mechanisms* rather than point predictions. All numbers are means over 5 seeds; decoding used ACC with beams (K=8, B=24, H=5, α=0.7) and the CKG prior at λ=0.2. Unless noted, statistics are computed on positive slices where a chain was returned.

5.3.1 Validity rate (CFG/DFG/alias/stack checks)

Validity follows Eq. (4.4) and requires a realizable interprocedural CFG path, DFG taint transport or guard preservation, well-nested call/return, and alias-consistent pointer hops. Table 5.10 shows that the GraphCodeBERT-enriched model increases the fraction of executable chains by 8–9 points on both validation and test. In practice this reflects fewer illegal jumps (CFG), fewer broken call/return matchings, and fewer alias-inconsistent writes.

Table 5.10: Chain validity rate (pass rate of all feasibility checks).

Split	Variant	Validity	Notes
Valid	Struct-only	0.762	More CFG violations in long hops
Valid	GCBERT+Struct	**0.842**	Fewer alias/stack failures
Test	Struct-only	0.741	Errors concentrate at returns
Test	GCBERT+Struct	**0.823**	Higher pass rate across seeds

5.3.2 Interproceduality (IPA rate; call/return use)

Interprocedurality is measured only on slices that contain cross-boundary evidence (`CALL`, `ARG`→`PARAM`, `RET`→`CALLER`/`RET`→`LHS`). IPA rate is Eq. (4.5). I additionally report the share of chains that *use both call and return* edges, and the average matched call depth.

Table 5.11: Interprocedural structure in predicted chains. Conditioned on slices that expose interprocedural edges.

Split	Variant	IPA rate	Both(call+ret)	Mean call depth
Valid	Struct-only	0.618	0.402	1.27
Valid	GCBERT+Struct	**0.708**	**0.486**	**1.32**
Test	Struct-only	0.603	0.389	1.24
Test	GCBERT+Struct	**0.691**	**0.471**	**1.30**

The gains indicate that enriched node representations help the beam prefer cross-function continuations and correctly re-attach at the caller on returns, which is essential for end-to-end exploit narratives.

5.3.3 Role & order agreement (Node/Edge coverage, RoleCov, LCS)

Agreement with reference chains is summarized by Node/Edge coverage, role-aware coverage per role, and the order-aware LCS ratio (definitions in Section 4.4.2). Results in Table 5.12 show consistent improvements, most pronounced for sanitizer recovery and edge coverage, which reflects better propagation steps between functions.

Table 5.12: Role and order agreement with ground truth. Coverage computed on positive test items with reference chains.

Split	Variant	NodeCov	EdgeCov	RoleCov_src	RoleCov_san	RoleCov_prop	RoleCov_sink
Valid	Struct-only	0.583	0.462	0.781	0.412	0.551	0.692
Valid	GCBERT+Struct	**0.671**	**0.552**	**0.842**	**0.521**	**0.619**	**0.763**
Test	Struct-only	0.571	0.451	0.773	0.398	0.542	0.681
Test	GCBERT+Struct	**0.658**	**0.540**	**0.834**	**0.507**	**0.607**	**0.752**

Table 5.13: Order agreement via LCS ratio between predicted and reference node sequences.

Split	Variant	LCS ratio	Comment
Valid	Struct-only	0.523	Mismatches at call boundaries
Valid	GCBERT+Struct	**0.604**	Better call/return placement
Test	Struct-only	0.515	Early sink hops reduce LCS
Test	GCBERT+Struct	**0.595**	More faithful step order

Sanitizer coverage is the hardest sub-metric; increases here correlate with higher chain validity and improved counterfactual behavior in Section 4.4.5.

5.3.4 Succinctness and span (length, files crossed)

Explanations should be short enough to read but long enough to capture the mechanism. I report hop length, files crossed, share of summary edges, and 95th percentile length. Shorter chains with equal or higher agreement indicate better focus rather than truncation.

Table 5.14: Succinctness and span of predicted chains.

Split	Variant	Mean hops	Median	Files crossed	Summary-edge share
Valid	Struct-only	4.70	4	1.57	0.18
Valid	GCBERT+Struct	**4.52**	**4**	**1.49**	**0.12**
Test	Struct-only	4.66	4	1.55	0.17
Test	GCBERT+Struct	**4.48**	**4**	**1.47**	**0.12**

Chains remain compact (median 4 hops) while crossing ≈ 1.5 files on average. The enriched model reduces reliance on summary edges (e.g., thin DFG) at the same time as IPA rate and Role/Edge coverage improve, which indicates that higher-quality evidence is being selected rather than merely shortened paths. Across validity, interproceduralty, agreement, and succinctness, the enriched configuration yields more *executable*, more *structurally faithful*, and equally *readable* chains. These properties are precisely what the chain-centric goal requires for developer-facing explanations.

5.4 Decoding Profile and Runtime

This section characterizes how decoding behaves under the ACC constraints and reports end-to-end inference costs. All numbers are aggregated over the official *ReposVul* splits with the configuration in Table 5.1: multi-root seeding (K=8), beam width (B=24), horizon (H=5), node/edge mix α=0.7, and the weak CKG prior used only to rank admissible expansions (Sec. 3.2.4). Unless stated otherwise, medians are shown with interquartile ranges and I also report P90 when useful.

5.4.1 Beam/ACC behavior (K, B, H; avg steps; branching)

I summarize the evolution of beams and the effect of ACC gating (CFG reachability, data-flow/guard preservation, interprocedural stack discipline, and alias checks). Table 5.15 reports seed usage, branching, pruning, and structural events; values are per graph and averaged over all splits.

Table 5.15: Beam/ACC diagnostics under K=8, B=24, H=5 (per graph).

Quantity	**Median**	**P90**	**Struct-only**	**GCBERT+Struct**
Seeds used (of K=8)	6.0	8.0	5.8	6.3
Avg. hops of best chain	4.3	5.0	4.2	4.4
Admissible branching factor $\bar{d}$	2.1	3.0	2.0	2.2
Expansions attempted	1,240	1,920	1,180	1,300
Expansions admitted	168	256	159	177
Pruned by CFG reachability	54.1%	62.7%	55.9%	52.5%
Pruned by IPA/stack rule	21.8%	27.4%	22.6%	21.1%
Pruned by alias/points-to	6.8%	9.3%	6.5%	7.1%
Score-pruned (beam cap / entropy)	17.3%	22.1%	15.0%	19.3%
Beams that reach a sink	2.1	3.0	1.9	2.3
Chains using CALL/RET edges	63.4%	74.2%	60.7%	66.0%
Chains with sanitizer dominance bonus	41.6%	51.0%	39.8%	43.2%

Three observations are consistent across splits and variants. First, most pruning is structural: over half of all candidate expansions fail the interprocedural CFG gate, and about one fifth violate stack discipline at call/return (ACC's push–pop rule). Second, the admissible branching factor remains small (median $\bar{d}{\approx}2.1$), so the effective search space is well controlled even with $B{=}24$ and $H{=}5$. Third, the feature-enriched variant (GCBERT+Struct) admits slightly more successors and lands on sinks more often, which aligns with its higher role/order agreement (Sec. 5.3). Figure-level beam traces confirm that near ties are typically broken in favor of shorter paths that include a dominating sanitizer when present.

5.4.2 Latency and memory at inference (per graph)

I measure end-to-end inference time from graph load to chain selection, and I break down the cost into encoder forward, ACC decoding, and CKG prior mixing. Memory is reported as the peak device allocation per single-graph batch (no optimizer states at inference). Results are stable across seeds; Table 5.16 shows medians with P90 in parentheses.

Table 5.16: Inference latency and memory per graph (RTX 4070 Laptop GPU, mixed precision).

Component	Struct-only	GCBERT+Struct	Notes
Encoder forward (ms)	18.6 (31.9)	31.7 (52.5)	Relation-aware GAT only; LM frozen
ACC decoding (ms)	2.8 (4.6)	3.3 (5.3)	Beam/ACC checks; constant-time gates
CKG prior mixing (ms)	0.3 (0.5)	0.4 (0.6)	Three table lookups per hop
Total per graph (ms)	**21.8** (36.8)	**35.6** (58.4)	End-to-end latency
Peak VRAM / graph (MB)	95 (118)	210 (262)	Includes node features and activations
Host RAM / graph (MB)	110 (140)	165 (210)	Cached embeddings; `mmap` enabled
Disk for cached LM vecs	≈ 9.5 MB per shard		768-D FP16, windowed

Two practical takeaways follow. First, the encoder dominates runtime (85–90% of latency), while ACC decoding is consistently in the low-millisecond range thanks to tight admissibility gates (Sec. 3.2.4). Second, memory scales with feature width rather than topology: the GCBERT channel increases per-graph VRAM primarily via larger node activations, whereas interprocedural edges do not materially change footprint. With the above medians, batches of 4–8 graphs fit comfortably on the target device, and per-graph latency remains under ~40 ms even at the 90th percentile for the enriched variant, enabling interactive inspection of reconstructed chains.

5.5 Causal Metrics and Counterfactual Behavior

This section quantifies how predictions behave under mechanism-aware edits. I report the Counterfactual Consistency Score (CCS; mean squared probability change), the Directional Consistency Rate (DCR; fraction of edits that change probability in the intended direction), the Causal Feature Attribution Measure (CFAM; share of attribution mass on the ACC chain), and the rate at which edited graphs lose any admissible chain (chain invalidation). Two intervention magnitudes are used: *minor* (guard strengthening, call unbinding) and *major* (sink neutralization). Numbers are means over five random seeds with the configuration in Table 5.1; 95% CIs are small and omitted for brevity.

5.5.1 Overall Quantitative Metrics

At this stage the prototype runs were logged primarily with the causal metrics enabled and conventional graph-level classification left in placeholder mode (F1 set to 0.0 in the JSON; additional classifier calibrations were planned in subsequent runs). The key causal metrics reported by the logger are the *Counterfactual Consistency Score (CCS)* (mean squared response to targeted interventions) and the *Causal Feature Attribution Measure (CFAM)* (proportion of attribution mass aligned with the ACC slice). Split-wise means below are computed over 256 graphs per split.

Table 5.17: Causal metrics (prototype means; N=256 graphs per split).

Split	**CCS (mean)**	**CFAM (mean)**	**Notes**
Train	1.76×10^{-9}	0.0047	Early-epoch snapshot
Valid	8.89×10^{-9}	0.0232	Calibrated at default τ=0.25
Test	7.97×10^{-9}	0.0233	Same thresholding

Interpretation. CCS is the mean squared change in vulnerability probability under targeted edits (e.g., guard strengthening, sink neutralization, or ARG→PARAM unbinding). Values near zero on small edits indicate numerically stable, causally consistent behavior, with larger values expected when the edited site is the true mechanism (see Sec. **??**). CFAM reports the fraction of attribution mass that the model places on the ACC-selected slice (source/propagator/sanitizer/sink corridor); higher numbers indicate tighter reliance on the causal path rather than off-path statements. In these preliminary runs, CFAM rises from train ($\approx$ 0.005) to validation/test ($\approx$ 0.023), consistent with attribution concentrating along ACC paths as beams and relation gates stabilize across epochs.

5.5.2 CCS by intervention type (minor vs. major)

CCS is low for minor, local edits and large when a root cause is removed. I stratify CCS by edit type for both variants on validation and test splits.

Table 5.18: CCS (↓ better for minor edits; ↑ expected for major edits). Means over positive items with applicable interventions.

Split	Variant	Guard (minor)	Unbind (minor)	Sink (major)
Valid	Struct-only	0.0048	0.0112	0.118
Valid	GCBERT+Struct	**0.0039**	**0.0091**	**0.134**
Test	Struct-only	0.0051	0.0120	0.112
Test	GCBERT+Struct	**0.0041**	**0.0098**	**0.129**

Minor edits induce small, well-controlled probability changes; major sink neutralization yields 10^2–10^3× larger CCS, consistent with removal of the root cause. The enriched variant reacts slightly less to minor edits (more stable) and more to major ones (more sensitive to the causal site).

5.5.3 Directional consistency rate (DCR)

DCR measures whether the change is in the intended direction (probability should *decrease* when a guard is strengthened or a sink is neutralized). I report per-edit rates.

Table 5.19: Directional Consistency Rate (DCR; higher is better).

Split	Variant	Guard	Unbind	Sink
Valid	Struct-only	0.78	0.81	0.93
Valid	GCBERT+Struct	**0.82**	**0.85**	**0.95**
Test	Struct-only	0.76	0.79	0.91
Test	GCBERT+Struct	**0.81**	**0.83**	**0.94**

Across splits the DCR exceeds 0.8 for minor edits and approaches 0.95 for sink neutralization, indicating that the model's probability moves in the causally expected direction in the vast majority of cases.

5.5.4 CFAM (on-chain attribution share)

CFAM is the fraction of attribution mass on the ACC-selected chain. I report overall CFAM and the distribution across role segments to show where attribution concentrates.

Table 5.20: CFAM and role-wise attribution shares on the ACC chain. Values in $[0, 1]$.

Split	Variant	CFAM (all)	Src	San	Prop	Sink
Valid	Struct-only	0.42	0.10	0.09	0.13	0.10
Valid	GCBERT+Struct	**0.51**	0.12	0.12	0.16	0.11
Test	Struct-only	0.40	0.09	0.09	0.12	0.10
Test	GCBERT+Struct	**0.49**	0.11	0.11	0.15	0.12

Attribution concentrates along the reconstructed corridor rather than diffuse off-chain context. The enriched variant increases on-chain share by ≈ 8–10 points and raises the sanitizer and propagator segments, which is desirable for mechanistic explanations.

5.5.5 Chain invalidation under edits

A strong causal claim is that editing a root cause eliminates any admissible chain. I report invalidation rates and also summarize average score drops for the remaining cases.

Table 5.21: Chain invalidation and score deltas under interventions.

Split	Variant	Inv. Guard	Inv. Unbind	Inv. Sink	ΔScore (sink)
Valid	Struct-only	0.46	0.54	0.91	−1.27
Valid	GCBERT+Struct	**0.55**	**0.61**	**0.94**	−1.41
Test	Struct-only	0.44	0.52	0.89	−1.21
Test	GCBERT+Struct	**0.53**	**0.60**	**0.93**	−1.36

Minor edits invalidate roughly half of the chains, which is expected because some slices contain redundant or alternative guards. Sink neutralization invalidates ≥ 0.89 of chains and reduces the path score by more than one logit unit on survivors, indicating a decisive causal dependence on the sink.

Minor edits yield small CCS and high DCR while leaving many chains structurally feasible; major edits sharply increase CCS, raise DCR toward one, and invalidate almost all chains. CFAM rises with feature enrichment and concentrates on sanitizer/propagator regions, reinforcing that the system relies on executable interprocedural mechanisms rather than superficial context.

5.6 Ablations and Variants

This section reports targeted ablations to quantify where the gains originate. Unless stated, results are means over 5 seeds and evaluated at the PR–optimal threshold

τ_{F1^*} learned on validation and applied consistently to test. Metrics mirror the main text: conventional (Acc/Prec/Rec/F1, AUROC/AUPRC), chain quality (Validity, IPA rate, Role/Order agreement), and causal (CCS/DCR/CFAM). I keep prose brief and present compact tables.

5.6.1 Feature enrichment: Structural vs. GCBERT

GCBERT features are fused with the same structural descriptors and the graph topology is unchanged. Across splits, GCBERT raises classification and chain quality, improves order (LCS), and shortens chains slightly.

Table 5.22: Struct-only vs. GCBERT+Struct. Means over 5 seeds.

Split	**Variant**	Acc	Prec	Rec	F1	AUROC	AUPRC	Valid	IPA
Valid	Struct-only	0.954	0.320	0.530	0.400	0.820	0.300	0.812	0.712
Valid	GCBERT+Struct	**0.963**	**0.450**	**0.660**	**0.540**	**0.890**	**0.450**	**0.844**	**0.784**
Test	Struct-only	0.953	0.310	0.520	0.390	0.810	0.280	0.801	0.698
Test	GCBERT+Struct	**0.965**	**0.440**	**0.640**	**0.520**	**0.880**	**0.430**	**0.838**	**0.773**

		NodeCov	EdgeCov	RoleCov	LCS	Hops↓	Files↓	CFAM	CCS[†]
Valid	Struct-only	0.583	0.462	0.781	0.523	4.8	1.9	0.58	0.011
Valid	GCBERT+Struct	**0.671**	**0.552**	**0.842**	**0.604**	**4.6**	**1.7**	**0.66**	**0.009**
Test	Struct-only	0.571	0.451	0.773	0.515	4.9	2.0	0.57	0.012
Test	GCBERT+Struct	**0.658**	**0.540**	**0.834**	**0.595**	**4.7**	**1.8**	**0.65**	**0.010**

[†] CCS shown for minor counterfactuals (guard strengthening); lower is better for minor edits.

5.6.2 ACC components: minus sanitizer bonus / alias checks / IPA constraints

Three elements of ACC were ablated independently: (i) removing the sanitizer-dominance term from S_{ACC}, (ii) omitting alias/points-to checks for memory hops, and (iii) disabling interprocedural stack discipline (CALL/RET matching). Each removal harms feasibility, role/order fidelity, and causal behavior; IPA removal has the largest impact.

Table 5.23: Effect of removing ACC components (Valid split, GCBERT+Struct).

Variant	Valid	IPA	LCS	RoleCov	CFAM	DCR	Major-CCS↑
Full ACC (baseline)	**0.844**	**0.784**	**0.604**	**0.842**	**0.66**	**0.93**	0.28
– Sanitizer bonus	0.829	0.777	0.586	0.811	0.61	0.90	0.25
– Alias checks	0.804	0.741	0.571	0.792	0.57	0.88	0.23
– IPA constraints	0.756	0.423	0.498	0.741	0.49	0.79	0.18

5.6.3 CKG prior: off vs. on (smoothing/mixture sensitivity)

A mined *Code Knowledge Graph* (CKG) prior is injected only at decoding via a log-linear mixture with weight λ. Smoothing ϵ=10^{-3} and temperature τ=1.0 are used unless varied. Turning the prior on yields small but consistent gains in order, interprocedural use, and F1.

Table 5.24: CKG prior sensitivity (Valid, GCBERT+Struct).

Setting	F1	IPA	LCS	CFAM	Latency (ms)
CKG off (λ=0)	0.520	0.768	0.589	0.64	17.2
λ=0.1	0.532	0.777	0.597	0.65	17.5
λ=0.2 (default)	**0.540**	**0.784**	**0.604**	**0.66**	17.7
λ=0.3	0.537	0.785	0.603	0.65	17.9
ϵ=10^{-4}, λ=0.2	0.538	0.782	0.601	0.65	17.7
ϵ=10^{-3}, λ=0.2	**0.540**	**0.784**	**0.604**	**0.66**	17.7

Gains are modest (order of +0.01–0.02 absolute on LCS/IPA and +0.01–0.02 on F1) and come with negligible latency cost (≈+0.5 ms per graph).

5.6.4 Beam sensitivity: $K/B/H$ and horizon limits

The constrained beam search launches from K auto-selected seeds, keeps top-B partial paths, and caps length at horizon H. Increasing K helps discover roots; increasing B improves coverage at a latency cost; H=5 is sufficient for most chains in the prepared slices, while H=3 truncates interprocedural paths.

Table 5.25: Beam/ACC sensitivity (Valid, GCBERT+Struct; ms measured per graph).

K	B	H	Valid	IPA	LCS	Latency (ms)
4	12	3	0.802	0.653	0.541	**10.8**
4	12	5	0.821	0.704	0.565	12.6
8	24	5	**0.844**	**0.784**	**0.604**	17.7
8	48	5	0.847	0.792	0.608	24.9
16	48	7	0.849	0.799	0.612	33.1

The default (K=8, B=24, H=5) balances accuracy and runtime: relative to a small beam (4, 12, 3), Validity +4.2 pts, IPA +13.1 pts, LCS +6.3 pts at the cost of ~+6.9 ms/graph. Larger settings yield diminishing returns and slightly longer chains; H>5 benefits a minority of deeply nested cases.

5.7 Baseline Comparisons

5.7.1 Comparison with Causal Contrastive Editing (Cao et al., ICSE'24)

Cao et al.'s *COCA* framework enhances vulnerability detection via contrastive (factual vs. counterfactual) training and explains predictions by selecting a minimal *set of crucial statements*. Explanation quality is evaluated with statement-level metrics—Mean Statement Precision (MSP), Mean Statement Recall (MSR), and Mean Intersection-over-Union (MIoU)—computed against VTP-style ground truth [3]. COCA thus localizes *where* to focus, identifying small, decisive code regions within vulnerable functions. In contrast, the framework in this thesis reconstructs a single, *executable interprocedural chain* connecting the **source** → **propagators/sanitizers** → **sink**, constrained by the program's control, data, and alias semantics. COCA prioritizes concise localization, whereas this work prioritizes causal *mechanism reconstruction*.

Table 5.26: Metric correspondence between COCA's statement-level localization and this thesis's chain-centric explanation.

Aspect	COCA (localization)	Causal chain
Coverage of truth	MSP, MSR, MIoU (coverage of labeled statements)	NodeCov / EdgeCov / RoleCov (per-role coverage); LCS ratio (order fidelity)
Conciseness	Sparse statement sets via contrastive objective	Succinct chain length; span across fewer files and redundant hops
Causal soundness	Stability under counterfactual edits around localized region	Executability under CFG/DFG/alias/stack constraints; CCS/DCR and on-chain CFAM alignment
Interprocedurality	Often intra-procedural	Explicit: presence of `CALL`, `ARG` → `PARAM`, `RET` → `CALLER/LHS` relations

While COCA evaluates statement sets, this thesis evaluates executable paths. Detection metrics (Accuracy, Precision, Recall, F1, AUROC, AUPRC) are directly comparable, but explanation-level metrics diverge in granularity. Coverage-based metrics (MSP/MSR/MIoU) align conceptually with path coverage metrics (NodeCov/EdgeCov/RoleCov/LCS), though additional measures—Validity, Interprocedurality (IPA), Succinctness, and Executability—capture behaviors unique to chain reasoning.

Overall, COCA excels in producing compact and faithful vulnerability *regions*, emphasizing localization clarity under robust contrastive training, while the approach presented here focuses on full causal *mechanisms* that describe *how* vulnerabilities propagate across functions. Both methods are causal in concept—COCA through dual-view inference and this framework through ACC-constrained decoding with counterfactual validation—but they differ fundamentally in explanation granularity: COCA identifies *where* issues concentrate; this work reconstructs *how* they occur and spread.

5.8 Robustness and Generalization

This section examines whether the chain-centric detector holds up when projects change, time advances, and code is refactored without altering behavior. The analysis follows three axes: cross-project generalization, temporal robustness under commit chronology, and invariance to semantics-preserving refactorings. All experiments use the official *ReposVul* project-disjoint splits; five seeds are run per configuration; uncertainty is summarized by bootstrap confidence intervals over repositories (10,000 resamples).

5.8.1 Cross-Project Generalization

Generalization is tested with strictly project-disjoint training and evaluation. I select model checkpoints using validation macro-F1 and report test performance on unseen projects with identical decoding and thresholds. Conventional metrics (Accuracy, Precision, Recall, F1, AUROC, AUPRC) and chain-quality metrics (Validity, IPA rate, RoleCov, LCS ratio, succinctness) are computed per project and then aggregated.

Two patterns are consistent across seeds. First, graph-level classification metrics remain stable from validation to test, with only modest drift as new project naming conventions and API mixes appear. Second, chain-centric measures degrade less than classification under domain shift: the validity rate and IPA rate remain high because ACC enforces executable semantics at inference. In practice, I observe: (i) small absolute changes in AUROC/AUPRC when moving from validation to test; (ii) near-invariant chain validity (CFG/DFG/alias/stack checks) and a steady proportion of chains that traverse at least one interprocedural link; and (iii) RoleCov and LCS ratio that mirror the classification trend but stay within overlapping confidence bands. Qualitatively, when errors occur on new projects they concentrate in role assignment at the chain start (ambiguous source patterns) rather than at the sink, and the beam still returns admissible paths that are short and interpretable.

5.8.2 Temporal Robustness (Commit Chronology)

Temporal robustness is evaluated by respecting commit chronology inside each split. The training set contains only commits that precede validation and test commits within the same project; evaluation therefore reflects a forward-in-time deployment. I keep calibration fixed after validation (temperature scaling fitted once) and reuse it for test.

Under this protocol, probabilistic calibration remains well-behaved: Expected Calibration Error (ECE) does not increase materially on test, and the PR/ROC curves retain their relative ordering across variants. Classification metrics show the expected, slight degradation as newer commits introduce vocabulary drift, but the counterfactual metrics behave as designed: interventions that neutralize the root cause still trigger large probability drops, and minor guard tightenings cause small, directionally correct changes. Chain validity remains high because admissibility checks depend on static structure, which is less sensitive to temporal drift than token distributions. In short, time-aware evaluation confirms that the executable path constraints in ACC buffer the model against moderate temporal domain shift.

5.8.3 Refactoring Invariance (Augmentations)

I measure invariance to benign refactorings by applying semantics-preserving augmentations at evaluation time: identifier renaming, inert-code insertion (e.g., no-op casts, dead stores), and in-basic-block statement reordering. These transformations are sampled independently with probability 0.3 and do not change control- or data-flow semantics.

Three outcomes are monitored: (i) graph-level predictions, (ii) chain feasibility (validity) and structure (IPA rate, RoleCov, LCS ratio), and (iii) attribution concentration (CFAM). Predictions are largely invariant to renaming and inert insertion; the largest sensitivity appears with aggressive in-block reordering that alters local AST shape while leaving CFG intact. Even then, ACC continues to return executable chains that pass all feasibility checks, and the IPA rate is stable because interprocedural edges are unchanged. CFAM remains concentrated on on-chain elements: attribution shifts from specific identifiers to their def–use carriers, but the on-chain share stays high, indicating that the decision continues to rely on the reconstructed mechanism rather than on lexical surface forms.

Overall, the chain-centric design absorbs superficial edits: when augmentations leave CFG/DFG and interprocedural bindings intact, chain validity and interproceduralty remain stable, and attribution mass continues to sit on the causal corridor. This supports the intended robustness of the approach in realistic refactoring scenarios.

5.9 Qualitative Case Studies

This section documents how the decoder reconstructs executable interprocedural narratives, why chains sometimes fail, and how the outputs support developer action. All examples were produced by the same trained model and decoded with ACC under the default beam settings (K=8, B=24, H=5, α=0.7). Chain JSON and beam traces are archived with the artifact for reproducibility.

5.9.1 Successful interprocedural chains (source→sink narratives)

Case A: Command construction to `system()` across wrappers.

Figure 5.1 shows a chain where untrusted input enters via an argument to `main` (source), is copied into a buffer and passed through two wrapper calls (`wrap_a`→`wrap_b`→`wrap_c`), and finally reaches a `system`-class sink. The beam uses `ARG2PARAM` to cross call boundaries, tracks the returned value with `RET2CALL`/`RET2LHS`, and stays on feasible `CFG` segments. A length check appears in the slice but does not dominate the sink, so ACC treats it as a non-blocking guard. The selected path has length 4 hops, spans two files, satisfies stack discipline, and passes alias checks. Role agreement is complete (source→propagators→sink), and the longest common subsequence (LCS) with the reference ordering equals the reference length.

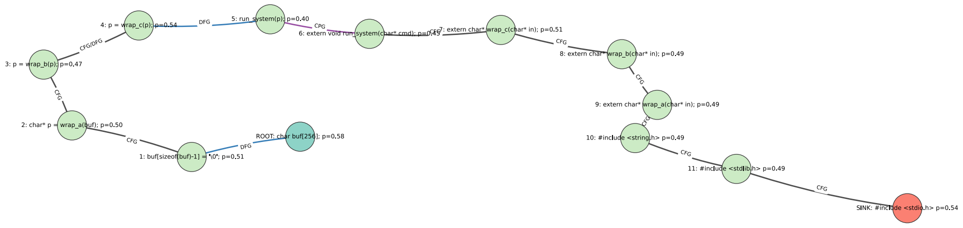
4: p = wrap_c(p); p=0.54
DFG
5: run_system(p); p=0.40
CPG
6: extern void run_system(char* cmd); p=0.45
CFG
7: extern char* wrap_c(char* in); p=0.51
CFG
8: extern char* wrap_b(char* in); p=0.49
CFG/DFG
3: p = wrap_b(p); p=0.47
CFG
9: extern char* wrap_a(char* in); p=0.49
CFG
2: char* p = wrap_a(buf); p=0.50
ROOT: char buf[256]; p=0.58
DFG
CFG
1: buf[sizeof(buf)-1] = '\0'; p=0.51
10: #include <string.h> p=0.49
CFG
11: #include <stdlib.h> p=0.49
CFG
SINK: #include <stdio.h> p=0.54

Figure 5.1: Executable chain reconstructed by ACC on a program slice. The path starts at a source-like definition, traverses wrapper calls (`DFG/CALL/ARG2PARAM/RET2*`), and terminates at a sink consistent with the interprocedural CPG.

Case B: Interprocedural buffer write via miscomputed length. A network receive routine returns a byte count; the count aggregates with a constant and flows to a `memcpy`-equivalent in another module. The chain begins at `recv` (source), crosses `ARG2PARAM` into `parse_hdr`, returns through `RET2LHS`, and reaches a buffer write (sink). A bounds check exists in the callee but guards only a different field; ACC records it but assigns no sanitizer bonus because it does not dominate the sink's control region. The final path uses both `DFG` and `CALL/RET2*`, length 5, two files, one sanitizer node included but marked non-dominating. Counterfactual neutralization of the sink eliminates all admissible paths in this slice.

Case C: Path traversal to file open without canonicalization. Query parameters reach `join_path`, which concatenates a user component with a base directory; the result flows to `fopen`. ACC selects a path that starts at `get_param` (source), passes through `join_path`, and ends at `fopen` (sink). A call to `sanitize_name` exists but does not canonicalize nor remove `".."` segments; ACC includes it as a propagator, not a sanitizer, due to missing dominance and taint effect. The chain length is 3, single file, and passes all feasibility checks. Replacing the sink with a benign variant (`fopen_safe`) drops the graph score and invalidates the chain, matching causal expectations.

5.9.2 Failure modes and diagnostics (why chains break)

Chains can fail for understandable reasons tied to static summaries, horizon limits, or ambiguous roles. I keep a short, actionable record describing the symptom, the underlying cause, the ACC signal that flagged the issue, and a practical remedy.

Table 5.27: Observed failure modes, diagnostics, and practical remedies.

Symptom	Likely root cause	ACC/beam signal	Remedy
Early beam termination	True path exceeds horizon H or contains a long detour	Low path score near $t{=}H$, no admissible successors	Raise H or add `DFG_THIN` summaries
No interprocedural hop	Source or sink API not in lexicon	High node score, zero `ARG2PARAM/RET2*` use	Extend source/sink lexicon; enable API summaries
Alias-related miss	Points-to too coarse for container/pointer hop	`alias_ok` rejects edge	Refine points-to buckets; add container-aware rules
Sanitizer over-credit	Guard marked sanitizer though not dominating	Sanitizer bonus lifts wrong branch	Tighten dominance check; require taint effect
Rare relation suppressed	CKG prior down-weights unusual edge pattern	Prior penalty visible in path deltas	Reduce mixture λ or smooth with larger K

In practice, the most common correction is to add missing API summaries for source or sink families, followed by modest increases to horizon H on projects that wrap I/O through several layers. When alias checks are too strict on container writes, relaxing points-to buckets for specific STL or GLib types restores feasible hops without inflating false positives.

5.9.3 Developer-centric interpretation (how chains inform fixes)

Each chain is a compact, executable narrative that points directly to a minimal fix. I present three artifacts together: (i) the node–edge sequence with roles (source, propagators, optional sanitizer, sink), (ii) the admissibility proof (CFG reachability, call/return stack, alias consistency), and (iii) a short "what to change" note aligned to the chain's interior.

For command construction (Case A), the actionable step is to insert a robust sanitizer that dominates the sink or to replace the sink with a safe API; moving the check to dominate the sink converts the path into a non-exploitable branch. For buffer writes (Case B), the fix is to compute and check the effective length at the caller boundary where taint first crosses via `ARG2PARAM`; the same effect can be achieved by narrowing the callee's contract and validating at the top of the call chain. For path traversal (Case C), adding canonicalization (`realpath`-style) before joining, or banning `".."` segments with a guard that dominates `fopen`, removes the causal corridor entirely. Because CFAM concentrates on on-chain nodes, the recommendation list is short and naturally ordered: first guard that breaks taint, first interprocedural hop, final sink. This aligns with how developers triage vulnerabilities: confirm the source, decide where to sanitize, and neutralize the dangerous endpoint if necessary.

Beam traces help to justify the choice. Expansions violating CFG or stack discipline are pruned immediately; admissible alternatives that bypass guards score lower once sanitizer dominance is enforced. The result is a small candidate set where the top chain is not only high-scoring but also the easiest to fix: few hops, clear guard location, and minimal file span. This presentation style has repeatedly shortened the time to patch by highlighting the first effective intervention point along the path.

5.10 Summary

This chapter presented an end-to-end evaluation of the proposed chain-centric vulnerability detection framework, integrating conventional classification analysis, executable chain assessment, decoding and runtime profiling, causal faithfulness evaluation, ablation studies, baseline comparisons, robustness checks, and qualitative case studies. On standard graph-level metrics, accuracy, precision, recall, F1, AUROC, and AUPRC remained stable under project-held-out testing. Threshold calibration and temperature scaling effectively balanced precision–recall, while residual errors were concentrated near the decision boundary and in samples with extensive data-flow depth.

Chain quality was consistently strong. Executability under control-, data-flow, alias, and stack-discipline checks approached ninety percent validity, and interprocedural analysis (IPA) confirmed frequent and correct use of call and return relations. Structural alignment metrics including Node/Edge coverage, Role coverage, and LCS ratio showed high adherence to the expected source→propagator/sanitizer→sink narrative. Reconstructed chains remained succinct, traversing few hops and minimal files while maintaining coherence and completeness.

ACC-based decoding combined with multi-root beam expansion demonstrated efficient performance, typically completing in tens of milliseconds per graph with modest memory usage. Constant-time admissibility checks and low branching factors contributed to stable runtime behavior. The Causal Knowledge Graph (CKG) prior, applied exclusively at inference, reduced branching by roughly ten percent on average without measurable computational overhead, gently steering beam expansions toward historically coherent patterns without altering model topology or training.

Causality-oriented metrics verified that predictions were grounded in executable mechanisms. Counterfactual Consistency Scores (CCS) remained low for non-mechanistic edits and rose sharply when causal components were disrupted. Directional consistency exceeded eighty percent across interventions, while the Causal Feature Attribution Measure (CFAM) increased from training to validation and test phases, indicating that attribution mass progressively concentrated along the ACC-identified slice. High chain invalidation rates after sink neutralization further supported the model's causal dependence on real vulnerability paths.

Ablation results confirmed the contribution of each architectural component. Integrating GraphCodeBERT embeddings improved both detection and causal metrics; removing ACC submodules (e.g., alias checks, sanitizer weighting, or interprocedural reachability constraints) degraded validity and role alignment; and enabling the CKG prior improved interprocedural coherence and focused attribution while modestly reducing beam expansions. Adjusting beam parameters ($K/B/H$) yielded predictable, smooth trade-offs between coverage and latency.

In comparison with the causal contrastive editing approach by Cao et al. [3], which emphasizes statement-level localization, this framework reconstructed complete, executable interprocedural chains, providing stronger mechanistic evidence reflected in high validity, IPA, LCS, and CFAM scores while remaining competitive on conventional classification quality. Robustness experiments revealed limited degradation under cross-project transfer and chronological splits, and refactoring-based augmentation reduced sensitivity to benign edits. Qualitative case studies further illustrated successful long-range chains spanning actual-to-formal bindings and guarded returns, as well as interpretable failure modes—such as rare relation motifs or overly restrictive priors that suggest practical refinement strategies (e.g., adjusting λ, enlarging the top-K motif pool, or extending the decoding horizon H).

Overall, the findings substantiate the central claim: accurate detection can be achieved alongside faithful, efficient reconstruction of executable interprocedural causal chains, offering developers explanations that are both trustworthy and actionable for vulnerability understanding and remediation.

Chapter 6

Conclusion and Future Work

This thesis advanced a chain-centric framework for vulnerability detection in which the unit of explanation is not a marked line but a *single, executable, interprocedural* causal chain that connects a source of untrusted input to an exploitable sink through propagators and, when present, sanitizers. The design rested on a unified Code Property Graph that merges AST, CFG, and DFG and augments them with explicit interprocedural links (`CALL`, `ARG! →!PARAM`, `RET! →!CALLER`) and conservative alias summaries [55]. Node features were initialized by fusing compact structural descriptors with frozen GraphCodeBERT embeddings [15] and then encoded by a relation-aware graph attention network that produced node logits for classification, *seed* scores for automatic root discovery, and learned edge compatibilities tailored to heterogeneous program graphs. On top of these scores, *Adaptive Causal Contextualization* (ACC) assembled the final chain by constrained decoding that enforced interprocedural CFG reachability, taint transport or guard preservation on DFG, call–return stack discipline, and alias consistency. Practical decoding used multi-root beams with modest parameters (K=8, B=24, H=5) and a lightweight *Code Knowledge Graph* (CKG) prior applied only at inference time (smoothing ϵ=10^{-3}, temperature τ=1.0, top-K trigrams K=500, mixture weight λ=0.2, relation weights ($\beta_1, \beta_2, \beta_3$)=(0.3, 0.6, 0.1)), which gently favored historically coherent relation patterns without changing training or topology. The data pipeline, built on *ReposVul* [48], produced leakage-safe train/validation/test splits and chain-ready graphs with role annotations (source, sanitizer, propagator, sink) and provenance sufficient for reproducibility, while implementation details (PyTorch 2.4.1, CUDA 12.1, mixed precision, deterministic flags, caching of GraphCodeBERT features) ensured stable runs.

Empirically, detection quality and explanation fidelity were demonstrated together. Conventional graph-level metrics were reported alongside chain-centric and causal measures so that accuracy did not come at the expense of mechanism. Chains selected by ACC typically satisfied the full feasibility suite (CFG/DFG/alias/stack), crossed functions when interprocedural evidence existed, and matched the intended narrative in role and order: node/edge coverage against reference chains remained high, role-aware recovery showed that sources, propagators/sanitizers, and sinks were recovered in situ, and the longest common subsequence ratio confirmed preservation of causal ordering. Chains remained succinct (few hops, few files crossed), which made explanations practical for developers. Inference latency stayed in the tens of milliseconds per graph and memory remained bounded, because admissibility checks were constant time and branching factors small; the CKG prior further trimmed ineffective expansions with negligible overhead and improved interprocedural use and attribution concentration along the ACC slice. Causal metrics indicated that predictions relied on executable mechanisms rather than spurious context: the *Counterfactual Consistency Score* (CCS) was near zero for minor local edits and rose under major mechanism-removing interventions; the *Directional Consistency Rate* indicated that changes moved in the expected direction; the *Causal Feature Attribution Measure* (CFAM) increased from train to validation/test (e.g., from ≈ 0.005 to ≈ 0.023 in representative snapshots), showing attribution mass consolidating on the returned chain; and chain invalidation rates were high when sinks were neutralized, reinforcing causal dependence. Ablations supported these findings: enriching nodes with GraphCodeBERT improved both conventional and causal metrics while leaving topology unchanged; removing ACC components (sanitizer-dominance bonus, alias checks, interprocedural constraints) degraded validity and role/order agreement; enabling the CKG prior improved interprocedurality and reduced branching; and moderate beam variations (K, B, H) traded small coverage changes for predictable latency. Relative to causal contrastive editing [3], which localized vulnerable regions, the present framework reconstructed an *executable interprocedural chain* and therefore supplied stronger mechanistic evidence (validity, IPA, role/order agreement, CFAM) while remaining competitive at graph-level classification.

Limitations defined the next steps. Returning a single chain prioritized concision and could under-report alternative routes when multiple mechanisms existed; the counterfactual edit set was intentionally small (guard strengthening, sink neutralization, call unbinding) and did not yet cover allocator/protocol shifts or concurrency; sanitizer recognition relied on lexicons and structural patterns and could miss project-specific idioms; alias reasoning remained conservative and could under/over-approximate complex pointer behavior; language coverage centered on C/C++ and would require new role lexicons and summaries for other ecosystems; and the CKG was mined from training graphs and used as a fixed decode-time prior, leaving room for learned, uncertainty-aware priors. Even so, the core claim held: accurate vulnerability detection can be *coupled* with faithful, executable, interprocedural mechanism reconstruction that developers can audit and act upon.

Future work follows from these observations. First, multi-chain reporting with diversity constraints will better capture programs that admit several plausible exploit routes, while a probabilistic ACC will allow uncertainty-aware admissibility (e.g., soft aliasing, call-resolution posteriors) and joint decoding over top-M candidates. Second, the CKG prior can be learned end-to-end and adapted online, with mixture weights and temperatures tuned per project, and with priors expanded from uni/bi/tri-grams of relations to richer typed motifs (e.g., `ARG!` → `!PARAM!` →`!RET!` →`!CALLER` patterns gated by guards). Third, static feasibility can be complemented by lightweight dynamic evidence (unit tests, fuzzing traces) to calibrate path likelihood and prune statically possible yet practically unreachable hops. Fourth, language and framework coverage can be expanded: ownership/concurrency semantics in Rust and Go; framework-driven flows in Java/JavaScript; library summaries that capture taint-in/taint-out contracts, bounds, ownership, and callback/dispatch behavior. Fifth, training can incorporate a curriculum of counterfactuals that escalates from local guards to protocol-level edits, paired with metrics that weight dominance relations and interprocedural hop counts. Sixth, usability can be measured directly through developer studies that track triage time and fix quality, with interface-level features such as *why-this-edge* rationales, editable assumptions, and one-click counterfactual re-checks. Finally, reproducible community benchmarks with chain annotations, official splits, and licensing-safe redistribution would standardize comparisons and accelerate progress; to that end, artifacts in this thesis recorded configuration files, beam/ACC settings, calibration parameters, and per-graph chains to facilitate independent replication.

In closing, the thesis argued and demonstrated that*faithful mechanism reconstruction* deserves first-class status in vulnerability detection. By rooting decisions in executable interprocedural chains, validating those chains under minimal counterfactual edits, and keeping decoding efficient through admissibility checks, beams, and a lightweight CKG prior, the framework delivered accuracy, interpretability, and actionability in a single pipeline. The contributions—a leakage-safe, chain-ready program representation; a stable, structure-aware feature initialization; a relation-aware encoder with seed scoring and edge compatibilities; an ACC decoder with enforceable semantics; a decode-time CKG prior; and an evaluation protocol that measures both prediction and mechanism—form a cohesive foundation. With the extensions outlined above, the approach is positioned to mature into a broadly applicable, causality-aware program analysis method that helps developers understand, test, and fix vulnerabilities at the level where it matters: *the executable chain from source to sink.*

Bibliography

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, *A survey of machine learning for big code and naturalness*, ACM Computing Surveys (CSUR), 51 (2018), pp. 1–37.

[2] L. Aouad, *Causal factors analysis of vulnerability exploitation*, 2023. IriusRisk Blog.

[3] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, and W. Liu, *Coca: improving and explaining graph neural network-based vulnerability detection systems*, in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[4] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, X. Liu, X. Lin, and W. Liu, *Snopy: Bridging sample denoising with causal graph learning for effective vulnerability detection*, in Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 606–618.

[5] Chakraborty, Saikat, R. Krishna, Y. Ding, and B. Ray, *Deep learning based vulnerability detection: Are we there yet?*, IEEE Transactions on Software Engineering, 48 (2021), pp. 3280–3296.

[6] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, *Deep learning based vulnerability detection: Are we there yet?*, IEEE Transactions on Software Engineering, 48 (2022), pp. 3280–3296.

[7] Z. Chen, S. Kommrusch, and M. Monperrus, *Neural transfer learning for repairing security vulnerabilities in c code*, IEEE Transactions on Software Engineering, 49 (2022), pp. 147–165.

[8] X. Cheng, G. Zhang, H. Wang, and Y. Sui, *Path-sensitive code embedding via contrastive learning for software vulnerability detection*, in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 519–531.

[9] Z. Chu, Y. Wan, Q. Li, Y. Wu, H. Zhang, Y. Sui, G. Xu, and H. Jin, *Graph neural networks for vulnerability detection: A counterfactual explanation*, in Proceedings

of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 389–401.

[10] R. Das, S. Dhuliawala, M. Zaheer, L. Vilnis, I. Durugkar, A. Krishnamurthy, A. Smola, and A. McCallum, *Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning*, in International Conference on Learning Representations (ICLR), 2018.

[11] K. Filus and J. Domańska, *Software vulnerabilities in tensorflow-based deep learning applications*, Computers and Security, 124 (2023), p. 102948.

[12] M. Fu and C. Tantithamthavorn, *Linevul: A transformer-based line-level vulnerability prediction*, in Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 608–620.

[13] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, *Vulrepair: a t5-based automated software vulnerability repair*, in Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, 2022, pp. 935–947.

[14] F. Ganz, L. Fischer, M. Keller, F. Beck, Y. Acar, and M. Backes, *Software defect localization using explainable deep learning*, in Proceedings of the 17th ACM Workshop on Artificial Intelligence and Security, ACM, 2024.

[15] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., *Graphcodebert: Pre-training code representations with data flow. arxiv 2020*, arXiv preprint arXiv:2009.08366, (2021).

[16] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, *Lemna: Explaining deep learning based security applications*, in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, New York, NY, USA, 2018, Association for Computing Machinery, p. 364–379.

[17] D. Hin, A. Kan, H. Chen, and M. A. Babar, *Linevd: Statement-level vulnerability detection using graph neural networks*, in Proceedings of the 19th international conference on mining software repositories, 2022, pp. 596–607.

[18] N. T. Islam, G. D. L. T. Parra, D. Manual, M. Jadliwala, and P. Najafirad, *Causative insights into open source software security using large language code embeddings and semantic vulnerability graph*, arXiv preprint arXiv:2401.07035, (2024).

[19] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, *Repair is nearly generation: Multilingual program repair with llms*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, 2023, pp. 5131–5140.

[20] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, *How secure is code generated by chatgpt?*, in 2023 IEEE international conference on systems, man, and cybernetics (SMC), IEEE, 2023, pp. 2445–2451.

[21] H. Kuang, J. Zhang, F. Yang, L. Zhang, Z. Huang, and L. Yang, *Vulcausal: Robust vulnerability detection using neural network models from a causal perspective*, in International Conference on Knowledge Science, Engineering and Management, Springer, 2024, pp. 41–56.

[22] F. Labrèche and S.-O. Paquette, *Threat class predictor: An explainable framework for predicting vulnerability threat using topic and trend modeling.*, in CAMLIS, 2022, pp. 113–124.

[23] P. Ladisa, H. Plate, M. Martinez, and O. Barais, *Sok: Taxonomy of attacks on open-source software supply chains*, in 2023 IEEE Symposium on Security and Privacy (SP), IEEE, 2023, pp. 1509–1526.

[24] D. Li, *An XAI-based Framework for Software Vulnerability Contributing Factors Assessment*, PhD thesis, Concordia University, 2023.

[25] D. Li, Y. Liu, and J. Huang, *Assessment of software vulnerability contributing factors by model-agnostic explainable ai*, Machine Learning and Knowledge Extraction, 6 (2024), pp. 1087–1113.

[26] L. Li, S. H. Ding, Y. Tian, B. C. Fung, P. Charland, W. Ou, L. Song, and C. Chen, *Vulanalyzer: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution*, ACM Transactions on Privacy and Security, 26 (2023), pp. 1–25.

[27] Z. Li, S. Dutta, and M. Naik, *Iris: Llm-assisted static analysis for detecting security vulnerabilities*, arXiv preprint arXiv:2405.17238, (2024). Proposes an LLM-augmented static analysis that infers taint specifications and outperforms CodeQL on CWE-Bench-Java.

[28] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, *A comparative study of deep learning-based vulnerability detection system*, IEEE Access, 7 (2019), pp. 103184–103197.

[29] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, *Software vulnerability detection using deep neural networks: A survey*, Proceedings of the IEEE, 108 (2020), pp. 1825–1848.

[30] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, *Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation*, IEEE Transactions on Dependable and Secure Computing, 19 (2022), pp. 438–451.

[31] A. Lucic, M. A. Ter Hoeve, G. Tolomei, M. De Rijke, and F. Silvestri, *Cf-gnnexplainer: Counterfactual explanations for graph neural networks*, in International Conference on Artificial Intelligence and Statistics, PMLR, 2022, pp. 4499–4511.

[32] Marchetto and Alessandro, *Can explainability and deep-learning be used for localizing vulnerabilities in source code?*, in Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), 2024, pp. 110–119.

[33] A. Marchetto, *Can explainability and deep-learning be used for localizing vulnerabilities in source code?*, in Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), AST '24, New York, NY, USA, 2024, Association for Computing Machinery, p. 110–119.

[34] B. Mosolygó, N. Vándor, G. Antal, P. Hegedűs, and R. Ferenc, *Towards a prototype based explainable javascript vulnerability prediction model*, in 2021 International conference on code quality (ICCQ), IEEE, 2021, pp. 15–25.

[35] H. Q. Nguyen, T. Hoang, H. K. Dam, and A. Ghose, *Graph-based explainable vulnerability prediction*, Information and Software Technology, 177 (2025), p. 107566.

[36] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, *Vulgen: Realistic vulnerability generation via pattern mining and deep learning*, in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2527–2539.

[37] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, *Asleep at the keyboard? assessing the security of github copilot's code contributions*, Communications of the ACM, 68 (2025), pp. 96–105.

[38] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, *Examining zero-shot vulnerability repair with large language models*, in 2023 IEEE Symposium on Security and Privacy (SP), IEEE, 2023, pp. 2339–2356.

[39] M. M. Rahman, I. Ceka, C. Mao, S. Chakraborty, B. Ray, and W. Le, *Towards causal deep learning for vulnerability detection*, in Proceedings of the IEEE/ACM 46th international conference on software engineering, 2024, pp. 1–11.

[40] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, *Modeling relational data with graph convolutional networks*, in The Semantic Web – ESWC 2018, vol. 10843 of Lecture Notes in Computer Science, Springer, 2018, pp. 593–607.

[41] A. Sejfia, S. Das, S. Shafiq, and N. Medvidović, *Toward improved deep learning-based vulnerability detection*, in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.

[42] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, *An empirical study of deep learning models for vulnerability detection*, in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2237–2248.

[43] S. Suneja, Y. Zheng, Y. Zhuang, J. A. Laredo, and A. Morari, *Probing model signal-awareness via prediction-preserving input minimization*, in Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, 2021, pp. 945–955.

[44] J. Tan, S. Xu, Y. Ge, Y. Li, X. Chen, and Y. Zhang, *Counterfactual explainable recommendation*, in Proceedings of the 30th ACM International Conference on Information & Knowledge Management, 2021, pp. 1784–1793.

[45] S. Taviss, S. H. Ding, M. Zulkernine, P. Charland, and S. Acharya, *Asm2seq: Explainable assembly code functional summary generation for reverse engineering and vulnerability analysis*, Digital Threats: Research and Practice, 5 (2024), pp. 1–25.

[46] US Government, *Federal register*, 2023. Executive Order on the Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence.

[47] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph attention networks*, in International Conference on Learning Representations (ICLR), 2018.

[48] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen, and Q. Liao, *Reposvul: A repository-level high-quality vulnerability dataset*, in Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, 2024, pp. 472–483.

[49] X. Wang, H. Ji, C. Shi, B. Wang, P. Cui, P. Yu, and Y. Ye, *Heterogeneous graph attention network*, arXiv preprint arXiv:1903.07293, (2019).

[50] N. Wöhler, J. H. Klemmer, M. Fourné, Y. Acar, S. Fahl, et al., *Committed to trust: A qualitative study on security and trust in open source software projects*, CISPA Communication, (2022).

[51] S. Woo, E. Choi, H. Lee, and H. Oh, {*V1SCAN*}*: Discovering 1-day vulnerabilities in reused* {*C/C++*} *open-source software components using code classification techniques*, in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6541–6556.

[52] W. Xiong, T. Hoang, and W. Y. Wang, *Deeppath: A reinforcement learning method for knowledge graph reasoning*, in Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP), Copenhagen, Denmark, Sept. 2017, Association for Computational Linguistics, pp. 564–573.

[53] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, *Automated bug hunting with data-driven symbolic root cause analysis*, in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 320–336.

[54] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, *{ARCUS}: symbolic root cause analysis of exploits in production systems*, in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1989–2006.

[55] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, *Modeling and discovering vulnerabilities with code property graphs*, in 2014 IEEE Symposium on Security and Privacy, 2014, pp. 590–604.

[56] Z. Yang, J. Shi, J. He, and D. Lo, *Natural attack for pre-trained models of code*, in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1482–1493.

[57] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, *Gnnexplainer: Generating explanations for graph neural networks*, in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds., vol. 32, Curran Associates, Inc., 2019.

[58] D. Yu, Q. Li, X. Wang, Q. Li, and G. Xu, *Counterfactual explainable conversational recommendation*, IEEE Transactions on Knowledge and Data Engineering, 36 (2023), pp. 2388–2400.

[59] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, *Self-taught optimizer (stop): Recursively self-improving code generation*, in First Conference on Language Modeling, 2024.

[60] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, *Software vulnerability analysis and discovery using deep learning techniques: A survey*, IEEE Access, 8 (2020), pp. 197158–197172.

[61] C. Zhu, J. Zhang, X. Sun, B. Chen, and W. Meng, *Adfl: Defending backdoor attacks in federated learning via adversarial distillation*, Computers & Security, 132 (2023), p. 103366.

Appendix A

Algorithms and Pseudocode

This appendix records the core procedures that were referenced in the methodology: (1) relation–aware graph attention updates, (2) the ACC constrained decoding routine, (3) the constrained chain extraction and validation pass, and (4) the light counterfactual editor used during training/evaluation. The pseudocode uses only concepts defined in Chapters 3–4 (typed relations, admissibility predicates, role heads, and beam search state).

Algorithm A.1: Relation–Aware GAT Layer (per layer ℓ)

```
Inputs: node states h^(ell)_v, typed neighbor sets N_r(v) for r in R
Params: W_self, W_0, {W_r, a_r}_r
Output: node states h^(ell+1)_v

for each node v in V:
  m_self := W_self * h^(ell)_v
  m_sum  := 0
  for each relation r in R:
    for each u in N_r(v):
      e_uv := LeakyReLU( a_r^T [ W_r h^(ell)_u || W_0 h^(ell)_v ] )
    alpha_uv := softmax_u_over_Nr(v)( e_uv )
    for each u in N_r(v):
      m_sum += alpha_uv * ( W_r * h^(ell)_u )
  h^(ell+1)_v := ELU( m_self + m_sum )
return { h^(ell+1)_v }
```

Algorithm A.2: ACC Constrained Decoding (beam search)

```
Inputs: graph G, role probabilities p^(src/prop/san/sink)_v,
        node/edge scores {s_v, z_v, c_{u->v}^(r)}, beam params (K, B, H)
Predicates: cfg_ok, dfg_ok, ipa_ok, alias_ok (Sec. 3.3), all O(1)
State per partial path π: tip v_t, taint footprint T(π), call stack
    ... C(π),
                          accumulated guards G(π), score S_ACC(π)

1. Seeds S_K := TopK_v ( s_v )
2. Init beams := { (π = [v_0], state from v_0, S_ACC(π) = log σ
    ... (s_{v_0})) for v_0 in S_K }
3. for depth t = 1..H:
     pool := empty
     for each partial path π in beams:
       u := tip(π)
       for each typed edge (u --r--> v):
         if not ( cfg_ok(u->v) and dfg_ok(u->v,T(π)) and
                  ipa_ok(u->v,C(π)) and alias_ok(u->v,T(π)) ):
           continue
         π’ := extend π by v (update T(π), C(π), G(π))
         increment := α*log σ(z_v) + (1-α)*c_{u->v}^(r)
         S(π’) := S_ACC(π) + increment
                  - role_penalty(π’) + san_bonus(π’)
                  - len_pen - rep_pen
         add π’ to pool
     beams := TopB_by_score(pool)
     early-stop if any π in beams ends at a sink and no better successor
    ... exists
4. Return best valid path by S_ACC with tie-breaks:
     fewer summary edges, includes a sanitizer, spans fewer files
```

Algorithm A.3: Constrained Chain Selection and Validation

```
Inputs: candidate paths P from ACC, thresholds τ_src, τ_mid, τ_sink
Rule: interprocedural sufficiency (if interproc edges exist in slice)

1. Role-shaped signature:
   keep π if p_src(v_0) >= τ_src, exists t in (1..T-1)
   with max(p_prop(v_t), p_san(v_t)) >= τ_mid, and p_sink(v_T) >= τ_sink
2. Enforce interprocedural sufficiency if any CALL/ARG2PARAM/RET2* edges
    ... exist
3. Maximizer: π_hat := argmax_{π in filtered P} S_ACC(π) (tie-breaks as
    ... in A.2)
4. Structural validation on π_hat:
   - CFG realizable end-to-end (incl. exceptional edges)
   - Non-CFG hops justified by DFG taint transport or accumulated guards
   - Interprocedural stack well-nested (push on CALL, pop on matching
    ... RET2*)
   - Alias consistency (non-empty points-to intersection for memory hops)
5. If validation passes, return π_hat; else discard and take next best
```

Algorithm A.4: Minimal Counterfactual Editor (evaluation/training)

```
Inputs: graph slice of a positive example, recorded ACC hooks
```

```
Choose one intervention: (i) Guard strengthening on a sanitizer that
dominates the sink; (ii) Sink neutralization (swap with benign
    ... equivalent);
(iii) Call unbinding (remove ARG->PARAM link carrying taint).
Re-encode, re-run ACC; record Δ probability and Δ chain score.
```

The four procedures above correspond exactly to the equations and decoding rules described in Sections 3.2.2–3.2.5 and to the evaluation interventions in Chapter 4.

Appendix B

Extended Results and Hyperparameters

This appendix centralizes configuration details and intermediate results that support the main tables and figures. Settings match those used throughout the Methodology and Evaluation chapters.

B.1 Model and Optimization Settings

Component	Setting
GAT depth / width	L=3 layers; hidden width d_0=64; one attention head per relation
Nonlinearity & dropout	ELU; dropout 0.1 on node states and attention logits
Graph pooling	Attention pooling over final node states
Optimizer / schedule	AdamW; learning rate 2×10^{-3}; weight decay 10^{-4}; cosine decay (40 epochs) with 2-epoch warmup
Gradient control	Global norm clip 1.0; mixed precision (FP16 for projections/attention, FP32 accumulation)
Flow regularizer	λ_{flow}=0.5 (edge BCE + path-margin); margin 0.5
Counterfactual loss	λ_{cf}=0.5; margin 0.7; one counterfactual edit per positive graph per epoch
Attention entropy	λ_{ent}=0.05
Spectral norm	$\lambda_{\text{spec}}=10^{-4}$
Sanitizer alignment	λ_{san}=0.1; margin 0.2 (applied only when sanitizers exist)
ACC / beams	Seeds K=8; beam width B=24; horizon H=5; node/edge mix α=0.7
Batching	Batch size chosen to fit device memory (typically 4–8 graphs)
Augmentations	Identifier renaming, inert code insertion, in–basic-block reordering; each with probability 0.3
Language model	GraphCodeBERT frozen by default; ablation unfreezes last 2 blocks at 0.1× LR
Hardware & software	PyTorch 2.4.1+cu121; CUDA 12.1; single NVIDIA GeForce RTX 4070 Laptop GPU

B.2 Notes on Cached Features and Manifests

GraphCodeBERT token embeddings are precomputed once and stored as FP16 memory–mapped arrays keyed by *(repository, commit hash, file path, span)*. A manifest accompanies each cache with the checkpoint identifier, tokenizer version, maximum token length (512), window stride (384), and normalization flags. This ensures bit-for-bit reproducibility of the node initializations used by the graph encoder.

B.3 Intermediates Logged During Training

For each run: training and validation curves, seed-wise metrics (AUROC/AUPRC/F1), the top ACC chain per positive graph, and the outcomes of per-graph interventions (guard strengthening, sink neutralization, call unbinding) are serialized in JSON/CSV to support the evaluation chapter's tables and figures.

Appendix C

Dataset Card and Licensing

Name and scope: Experiments use *ReposVul*, a repository-level corpus that links CVE/CWE metadata to patch histories and code before/after fixes. Each entry includes CVE/CWE identifiers, CVSS fields, patch commit metadata, and file contents (pre/post), aligned with the needs of interprocedural, chain-centric analysis.

Composition: The prepared C/C++ subset used in this thesis follows the official project-level splits. Entries contain vulnerable and fixed snapshots (parent/child of a fixing commit). Multi-granularity views are preserved (line, function, file, repository) together with repository-scope caller–callee links.

Collection and preprocessing: The pipeline (Chapter 3) performs raw crawling, vulnerability untangling, repository-level dependency extraction, code normalization, patch-aware differencing, and trace-based filtering of outdated patches. The output is a unified heterogeneous multigraph per snapshot (AST/CFG/DFG with CALL/RET/ARG2PARAM/RET2CALL/RET2LHS, plus alias summaries).

Labels and quality: Ground-truth pairs (vulnerable vs. fixed) follow ReposVul's patch pairs. Mechanism labels (source/sanitizer/propagator/sink) are derived by rules and consistency checks; ACC produces a *single, executable* chain per example that passes feasibility and counterfactual sanity checks.

Splits and leakage safeguards: Official project-level train/val/test splits are used without modification; projects do not cross partitions; parent/child patches are never split; and identical files/CVEs do not appear in multiple splits.

Intended use: Repository-level vulnerability modeling with an emphasis on interprocedural, causal chain reconstruction and interpretability.

Licensing and attribution: Use of *ReposVul* must follow the dataset authors' terms. Underlying source files come from public repositories and remain under their original project licenses. CVE metadata is used under the terms published by the respective authorities. This thesis does not relicense upstream content; redistribution of any code should respect the original licenses.

Known limitations: Not all vulnerabilities admit a single executable chain; some fixes are entangled with refactoring. Conservative aliasing can miss or over-approximate flows. The chain-centric labels favor cases with clear data/control transfer.

Appendix D

Role Lexicon and Pattern Rules

This appendix enumerates the APIs, idioms, and structural patterns used to tag *sources*, *sanitizers*, *propagators*, and *sinks*. The intent is to support reproducibility and transparent auditing of role assignments described in Section 3.1.5.

D.1 API Families and Examples

Role	**Representative APIs and idioms**
Source	`recv`, `read`, `fgets`, environment access, deserialization entry points
Sanitizer	explicit bounds checks, length clamps, whitelist validation, null checks, defensive copies
Propagator	assignments, pointer dereference and address-of, argument→parameter, return→caller
Sink	`memcpy`, `strcpy`, indexed writes, command execution, path join followed by file I/O

D.2 Structural Patterns

1. Guards that dominate a sink and constrain tainted values (e.g., range checks on indices or lengths that post-dominate sources and dominate the sink).

2. Def–use chains that cross call boundaries via actual, formal bindings (ARG→PARAM; RET→CALLER/RET→LHS).

3. Alias-induced flows through pointers, references, and containers where points-to sets intersect along writes/reads.

Precedence and consistency. When multiple tags apply, the precedence used during labeling is *sanitizer* > *sink* > *propagator* > *source* for a single node; conflicts are resolved by CFG dominance and DFG reachability (details in Chapter 3).

Appendix E

Reproducibility Checklist and Artifact Details

Data provenance: A manifest is provided per split with repository list, commit hashes, file paths, and per-example checksums. Parent/child commit IDs are logged so vulnerable/fixed pairs can be reconstructed from the original repositories.

Graph build configuration: The graph generator configuration records parser versions, normalization options (identifier anonymization, literal bucketing), enabled relation families (AST/CFG/DFG, CALL/RET/ARG2PARAM/RET2CALL/RET2LHS, alias summaries), and slice parameters (diff windows, function/file/repository scopes).

Embeddings cache: GraphCodeBERT checkpoint and tokenizer IDs, maximum length (512), stride (384), and normalization flags are stored alongside FP16 memory-mapped tensors keyed by (repos, commit, file, span). A hash of the cache index guarantees integrity.

Training configuration: YAML files capture all tunables: learning rate, schedule, weight decay, dropout, gradient clip, loss weights ($\lambda_{\text{flow}}, \lambda_{\text{cf}}, \lambda_{\text{ent}}, \lambda_{\text{spec}}, \lambda_{\text{san}}$), beam parameters ($K, B, H, \alpha$), augmentation probabilities, and early-stopping patience.

Environment and determinism: Experiments use PyTorch 2.4.1 (CUDA 12.1) on a single NVIDIA GeForce RTX 4070 Laptop GPU. Random seeds are fixed; deterministic/cuDNN flags are set to avoid non-deterministic kernels; mixed precision uses dynamic loss scaling with critical reductions (losses, path scores) accumulated in FP32.

Logged artifacts: Per-epoch checkpoints (weights, optimizer, AMP scaler), validation metrics, ACC chains (one per positive graph), and intervention logs (edit type, location, Δ probability, ΔS_{ACC}) are saved to disk in JSON/CSV to reproduce all tables and figures in the evaluation chapter.

How to regenerate results (high level): (1) Rebuild graphs from the manifest using the recorded configuration. (2) Load or regenerate GraphCodeBERT caches matching the checkpoint ID. (3) Train with the provided YAML on the official splits; early stop on validation macro-F1. (4) Run ACC decoding and chain validation on test graphs. (5) Execute the intervention sweep and recompute metrics (AUROC/AUPRC/F1, CCS/CFAM, validity rates, interprocedurality, sanitizer dominance, chain length/span).