Detecting Interprocedural Software Vulnerabilities Using Causal Deep Learning

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
January 2026

Abstract

Modern software systems often exhibit long interprocedural execution paths that span multiple functions and modules. The resulting interactions can introduce subtle vulnerabilities that adversaries may exploit. Many existing static, dynamic, and learning-based vulnerability detectors operate as coarse function, file or graph-level classifiers and provide limited insight into how a vulnerability originates and propagates through the program. This dissertation introduces a unified approach that integrates vulnerability detection with explicit reconstruction of executable root-to-sink vulnerability traces, leveraging taint-propagation analysis to produce structured, traceable explanations that support exploitability assessment and remediation in real-world software systems. The approach represents programs as augmented graphs that integrate syntactic and semantic relations within and across functions. Interprocedural program representations are initialized via a pretrained structure-aware encoder and subsequently refined by a relation-aware graph neural network that integrates adaptive causal reasoning guided by Causal Knowledge Graph priors. Grounded in this representation, a constrained decoder constructs the most probable executable root-to-sink chain for each detected vulnerability and validates semantic consistency and aliasing constraints against program dependencies. Experiments evaluate detection performance and explanation quality using standard detection metrics combined with causal and propagation-path criteria. These criteria assess whether predicted chains satisfy feasibility constraints, recover the relevant interprocedural structure, and remain stable under targeted code edits. Experimental results demonstrate that the approach consistently generates accurate, executable root→propagation→sink chains that reflect underlying interprocedural behaviour and provide structured explanations for effective vulnerability localisation and root-cause analysis in complex software systems.

Acknowledgments

I am profoundly thankful to the Almighty for granting me the strength and perseverance to complete this dissertation. I am deeply grateful to my supervisor, Dr. Yasir Malik, for his guidance and support. His mentorship has been invaluable throughout this research. I also wish to express my heartfelt gratitude to my family, whose constant love, encouragement, and understanding have been essential in completing this paper.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern digital infrastructure, financial services, scientific workflows, and everyday communication all depend on large, evolving software systems. As these systems scale in size and heterogeneity, codebases often span millions of lines of code and are developed by distributed teams over long periods. This growth in complexity increases architectural coupling and the likelihood that subtle defects persist into production. Unlike general software bugs, software vulnerabilities are defects that an adversary can exploit to violate the confidentiality, integrity, or availability of software systems, and their effects may propagate across dependent components and services [30, 31, 42].

Software vulnerability detection remains a critical yet complex challenge in software engineering. 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. Accordingly, rigorous vulnerability analysis centers on identifying which root→propagation→sink chains remain feasible under these constraints, rather than treating defects in isolation. This work therefore treats vulnerabilities as root→propagation→sink mechanisms to be reconstructed and validated end to end. In this context, revealing these causal chains in practice demands techniques that can approximate program behaviour before runtime.

Traditional vulnerability detection techniques provide important, but incomplete, coverage of this problem space. *Static analysis* examines code without executing it, relying on rule-based systems, type reasoning, data-flow frameworks, and abstract

interpretation to infer program behaviour at scale [8, 24, 33]. *Dynamic analysis* executes programs with concrete or instrumented inputs to observe runtime behaviour directly, often using monitoring, tracing, or sandboxing to detect anomalous states [6, 40]. In practice, these approaches are complemented by *fuzzing* and penetration testing, which attempt to exercise programs under diverse or adversarial inputs in order to trigger faults and expose exploitable conditions. Static analysis offers broad coverage but tends to over-approximate feasible execution paths, producing false positives and limited insight into concrete exploitability, while dynamic analysis, fuzzing, and penetration testing provide high-fidelity traces but often struggle with input coverage, scalability, and reproducibility on large, modular systems [30, 31].

In recent years, software vulnerability detection has increasingly shifted toward data-driven approaches, including machine learning and deep learning models, which aim to automate the discovery of vulnerabilities by learning patterns from code representations. Despite their promise, deep learning models often rely on statistical correlations in training data, capturing superficial patterns rather than the underlying mechanisms that give rise to vulnerabilities. Consequently, these models tend to focus on isolated statements or small subgraphs that are predictive of vulnerability labels, emphasizing where a flaw is likely to occur rather than how it emerges and propagates through interacting control and data flows [31, 42]. This correlation-centric bias introduces several limitations: models may become sensitive to dataset-specific artifacts such as naming conventions or formatting and may fail under benign transformations, including refactoring, formatting changes, or identifier renaming [31, 42]. Accurately modeling vulnerability behavior therefore requires reasoning not only about local syntax and intra-procedural structure but also about interprocedural semantics, encompassing call and return edges, argument-to-parameter bindings, return-to-caller relationships, and conservative aliasing across functions and module boundariess [30]. Existing vulnerability detection techniques-including static and dynamic analysis, fuzzing, penetration testing, and modern data-driven detectors-thus provide important yet incomplete coverage of these interprocedural behaviors and remain insufficient to capture the full causal structure of real-world vulnerabilities. Classical static, dynamic, and fuzzing-based solutions are effective at finding local defects or triggering specific failure scenarios, but they rarely find interprocedural vulnerability. Similarly, most data-driven detectors are trained to classify localized code regions or limited graph neighborhoods and tend to focus on point wise predictions of *where* a vulnerability may occur, rather than modeling *how* vulnerable state propagates across functions and modules in a causally meaningful way [30, 31, 42].

In real software systems, vulnerabilities often arise from complex interactions that span multiple functions, modules, or files. Tainted input may flow through several layers of wrapper functions before reaching a sensitive sink; resource management errors may involve initialization in one component and misuse in another; and

access-control checks may be applied inconsistently across different call paths. These behaviours are further complicated by the use of shared libraries, callbacks, framework conventions, and configuration-driven control flow, where the relevant control and data dependencies are distributed across APIs, modules, and sometimes even language boundaries. As a result, the conditions that make a defect exploitable often depend on long, cross-cutting chains of calls and data transformations that are only partially visible in any single local context. Existing static, dynamic, fuzzing, and data-driven techniques typically approximate these patterns using limited context or heuristics [30, 31].

To address these challenges, this work adopts a *causal and context-aware* perspective on vulnerability detection. In this approach, *context awareness* refers to the capability of a model to understand how program elements relate not only syntactically, but also semantically, through control flow, data dependencies, and functional interactions [11, 41]. Vulnerabilities often do not originate from a single isolated statement; instead, they emerge from the interplay of multiple operations over time and across function boundaries, so capturing this semantic context is essential for reconstructing meaningful explanations of vulnerability behaviour, especially in interprocedural scenarios [30]. At the same time, the dissertation builds on principles from *causal inference*, which seek to distinguish true cause–effect relationships from spurious correlations and have motivated recent causal and explainable approaches to vulnerability detection [4, 7, 15, 29]. In the context of vulnerability detection, the integration of causal reasoning within program analysis is both strategically advantageous and fundamentally essential. Effective remediation hinges on accurately identifying the specific code elements and propagation pathways that are causally responsible for vulnerabilities. Accordingly, this work adopts an approach that develops a context-aware, data-driven causal framework to reconstruct interprocedural root-to-sink vulnerability propagation paths within source code. This approach seeks to harmonize vulnerability prediction with executable, causally grounded explanations, elucidating not only the presence of vulnerabilities but also the precise manner in which they emerge and disseminate across functions and modules. Such causally informed interpretations provide a necessary foundation for trustworthy, actionable vulnerability diagnostics and remediation guidance.

1.1 Research Gaps and Objectives

Software vulnerability detection has advanced considerably over the past decade, yet critical gaps remain in both data-driven and interprocedural analysis techniques. Causal deficiency and generalization constitute a major limitation: many data-driven detectors rely on superficial statistical correlations rather than capturing the underlying mechanisms that produce vulnerabilities, resulting in models that are sensitive to benign code refactorings and exhibit poor cross-project generalization [31, 42]. Interprocedural reasoning and chain reconstruction represent a second

significant gap. Most existing approaches focus on individual functions or limited intra-procedural slices, neglecting interprocedural dependencies and thereby failing to reconstruct executable root-to-sink vulnerability propagation chains across large, modular software systems [30]. Addressing these gaps requires representations and reasoning mechanisms that faithfully model control and data dependencies across functions, files, and modules, enabling systematic construction of viable and minimal causal chains. Accordingly, this research pursues two interrelated research objectives.

Objective 1: Design of a Program Representation Capturing Interprocedural Control and Data Dependencies. This research designs a program representation that captures interprocedural control and data dependencies by integrating Abstract Syntax Trees (ASTs), Control-Flow Graphs (CFGs), Data-Flow Graphs (DFGs), and Program/Code Property Graphs (PDG/CPG), producing a unified model suitable for root-to-sink causal analysis.

Objective 2: Development of a Constraint-Based Approach for Generating Executable Interprocedural Vulnerability Chains. This work develops a constraint-based approach for generating executable interprocedural vulnerability chains, explicitly modelling control flow, data dependencies, and aliasing to ensure that the resulting chains are *executable and semantically consistent*. These objectives guide the methods developed in this dissertation and form the basis for the contributions outlined in the following section.

To achieve these objectives, real-world vulnerable repositories are parsed into ASTs, CFGs, and DFGs, which are then integrated into a heterogeneous program/Code Property Graph augmented with interprocedural semantics, including call/return edges, argument-to-parameter mappings, return-to-caller links, and conservative aliasing. Nodes in this graph are encoded using a structure-aware pretrained model (GraphCodeBERT) combined with structural features, and embeddings are further refined by a graph-based encoder equipped with attention mechanisms, Adaptive Causal Contextualization (ACC), and a Causal Knowledge Graph (CKG) prior [46]. On top of this encoder, a constrained beam search decoder systematically explores feasible control- and data-flow paths to assemble executable root-to-sink chains that are consistent with interprocedural semantics, enabling reconstruction of the causal mechanisms underlying real-world software vulnerabilities.

1.2 Contributions and Significance

This research transforms vulnerability explanation as an interprocedural causal chain. It links root causes to exploitable sinks. The approach integrates static and dynamic analysis, fuzzing, penetration testing, and learning-based detection. It

traces how untrusted input enters the system and travels through control and data dependencies. It shows how sanitization is bypassed and leads to a vulnerable operation. The method turns opaque model outputs into interpretable evidence. These chains localize vulnerabilities and provide actionable remediation insights. Overall, this work advances causality-aware vulnerability analysis with improved transparency and utility.

C1 - Chain-Centric Interprocedural Program Representation. This work extends the code property graph by integrating explicit interprocedural semantics-call/return edges, argument-to-parameter mappings, return-to-caller links, and conservative aliasing-to preserve control and data dependencies across functions and files. This unified representation supports the construction of executable root→propagation→sink chains.

C2 - Interprocedural Causal Vulnerability Chain Construction. A context-driven causal process generates explicit interprocedural root-to-sink vulnerability chains. Real-world vulnerable repositories are parsed into ASTs, CFGs, and DFGs, which are integrated into a heterogeneous Program/Code Property Graph. Nodes are encoded using GraphCodeBERT combined with structural features, and embeddings are refined with a graph-based encoder equipped with multi-head attention, Adaptive Causal Contextualization (ACC), and a Causal Knowledge Graph (CKG) prior. A constrained beam-search decoder systematically enumerates feasible interprocedural control-flow and data-flow paths, enforcing aliasing and call/return consistency, and assembles *executable, minimal, and semantically consistent* vulnerability chains.

1.3 Thesis Outline

This research is structured as follows. Chapter 1 introduces the problem context, articulates the research gaps, formulates the research questions and objectives, and summarizes the main contributions. Chapter 2 reviews related work in the software vulnerability detection domain. Chapter 3 describes the methodology, including dataset preparation, interprocedural graph construction, chain-centric representation, and the model architecture with its decoding pipeline. Chapter 4 presents the experimental setup, and detailed results focusing on detection accuracy, chain quality, causal analyses, robustness, and case studies. Chapter 5 concludes with key findings, implications, limitations, and future research directions. Appendices provide supplementary materials such as algorithms, model equations, metrics, extended results, and reproducibility details.

Chapter 2

Literature Review

The growing need for more reliable and interpretable software vulnerability detection methods has driven extensive exploration of deep learning (DL) based approaches in recent research. These methods aim to automate the identification of vulnerabilities by learning patterns from code representations, often outperforming traditional static and dynamic analysis. However, despite their success, deep learning models still face challenges in scalability, generalizability, and transparency. To address these issues, researchers have investigated both the refinement of DL-based detection techniques and the development of methods to enhance model explainability and causal reasoning capability. This pursuit of more robust and interpretable models leads directly to the exploration of advanced DL architectures and explainability techniques discussed in this chapter.

2.1 Related Works

2.1.1 Static and Dynamic Analysis Techniques

Static analysis examines software artifacts without executing them and seeks to reason soundly or semi-soundly about all possible program behaviours. Classical approaches rely on lattice-based data-flow frameworks, abstract interpretation, and type- or effect-based reasoning to approximate control and data dependencies at scale [5, 41]. Their precision depends on sensitivities to control flow, calling context, execution paths, and heap or field abstractions. Interprocedural static analysis, in particular, requires accurate call-graph construction, supported by techniques such as Class Hierarchy Analysis, Rapid Type Analysis, and points-to analysis to resolve aliasing and dynamic dispatch [1, 22]. These analyses operate over canonical program representations: Abstract Syntax Trees (ASTs) encode syntactic structure, Control-Flow Graphs (CFGs) capture feasible execution order and branching, Data-Flow Graphs (DFGs) trace definition–use chains of variable values, and Program Dependence Graphs (PDGs) unify data and control dependencies to support slicing

and taint tracking. The Code Property Graph (CPG) integrates AST, CFG, and DFG into a heterogeneous multigraph and has been widely adopted to support scalable, interprocedural vulnerability queries from sources to sinks [41].

Despite their ability to detect potential issues early, classical static analyses routinely suffer from false positives due to conservative over-approximation of sanitization logic, aliasing, reflection, and dynamic dispatch [8, 24]. Comparative studies of widely used static analyzers, such as Flawfinder, RATS, Cppcheck, SpotBugs, and PMD, report substantial variability in detection rates and false-positive profiles across languages and vulnerability categories [14]. These limitations have motivated integration of machine learning and deep learning into static analysis pipelines. For example, IRIS combines large language models (LLMs) with static analysis to infer taint specifications and expand coverage of vulnerability patterns while reducing false discoveries, outperforming rule-centric tools such as CodeQL on several benchmarks [20]. Other hybrid approaches use ML classifiers to post-process static warnings, filter likely false positives, or prioritise alerts for manual inspection, thereby improving scalability and analyst productivity [9]. Static analysis thus remains a foundational component that provides structured, semantically rich program views to downstream learning-based detectors, even as its limitations on complex, rapidly evolving, and domain-specific codebases remain an open challenge [1, 5].

Dynamic analysis complements static reasoning by monitoring concrete or symbolic program executions to detect vulnerabilities at runtime. Representative techniques include fuzzing, which automatically mutates inputs to explore diverse execution paths; runtime sanitizers, which instrument code to detect memory safety violations and undefined behaviour; and symbolic or concolic execution, which uses constraint solvers to systematically explore feasible paths leading to potential faults [39, 40]. Dynamic methods produce high-fidelity witness traces that are directly actionable for debugging and exploit validation, but they are constrained by path explosion, environment modelling issues, and limited input or configuration coverage, especially in large-scale interprocedural systems [40]. To mitigate these limitations, recent work has proposed hybrid static–dynamic workflows in which machine learning and reinforcement learning guide fuzzing campaigns, prioritise target functions, or learn path selection heuristics from execution feedback, thereby improving coverage and discovery rates [34, 38]. Overall, static analysis offers scalability and whole-program coverage, whereas dynamic analysis provides precise execution evidence; their integration, increasingly mediated by data-driven techniques, forms a central strand in contemporary research on automated software vulnerability detection.

2.1.2 Deep Learning Based Techniques

Deep learning is reshaping many areas of computer science, and graph neural networks (GNNs) have rapidly shown strong performance in software vulnerability detection [5, 12, 22, 45]. By modeling code as graphs whose nodes represent program elements and edges encode syntactic and semantic relations, GNN-based models can capture complex software dependencies and localize vulnerable statements, for example in the line-level setting demonstrated by Hin et al. [12]. However, many of these models operate as opaque "black boxes," making it difficult to understand why a particular code fragment is classified as vulnerable [19, 25], even when prediction accuracy is high. In security-critical contexts, this lack of transparency is problematic and developers must understand *why* a vulnerability is reported in order to implement appropriate fixes, and recent policy initiatives, such as the U.S. Executive Order on safe, secure, and trustworthy AI, explicitly emphasize the need for reliable and explainable AI behaviour in high-stakes systems [35].

Zhou et al. [21] conducted a systematic empirical study of design factors that influence the performance of deep learning based vulnerability detectors. They constructed two datasets capturing both data and control dependencies for 126 vulnerability types and implemented a Joern based detection pipeline to enable controlled comparisons. Their experiments quantified the impact of several key choices, including techniques for handling class imbalance, the inclusion of control dependence information in code representations, and the selection of neural network architectures, on downstream detection accuracy. The results provide practical guidance for building effective DL based vulnerability detection systems, but the work remains correlation oriented and does not explicitly incorporate causal deep learning or formal causal reasoning principles.

Li et al. [31] 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 authors 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 authors urged more investigation on code patterns, possible inclusion of causal detection for better generalization, and addressing of the shortcomings of present model interpretation techniques.

2.1.3 Causal Reasoning and Causal Deep Learning Techniques

Conventional deep learning based vulnerability detectors are highly sensitive to spurious correlations and dataset shift. VulCausal, presented by Kuang et al. [15], introduces a causal perspective by defining a structural causal model over user-defined identifiers, API library calls, and code structure, then applying backdoor-style adjustments during reasoning to suppress confounding and remove misleading associations. This yields more stable and accurate vulnerability predictions than correlation-driven baselines, but the approach still faces practical limitations, including scalability to very large codebases, limited language coverage, and the absence of explicit executable causal chains beyond improved metrics. More broadly, such methods highlight both the promise of causal inference for vulnerability detection and the substantial modelling effort required, in line with ongoing work on learning more robust, causally grounded optimizers and representations [44].

Le et al. [30] examined how existing deep learning based detectors handle vulnerabilities that span multiple basic units of code (MBUs) and showed that models trained and evaluated at the individual basic unit (IBU) level systematically struggle with such cases, overstating accuracy by focusing on isolated units. Although the work does not employ explicit causal reasoning, it provides a detailed characterization and taxonomy of MBU vulnerabilities and analyses their incidence and detection accuracy in modern DL based detectors. The main contribution is a framework for correctly integrating MBU vulnerabilities into DL based detection pipelines, underscoring the need to evaluate models across a spectrum of granularities, which is particularly important in contemporary multi file, multi module software systems. Complementary efforts, such as Nong et al. [27] on authentic vulnerability generation via pattern mining and deep learning, and Woo et al. [37] on identifying 1 day vulnerabilities in reused open source components, further refine understanding of vulnerability types and their prevalence in real world code.

Cao et al. [4] introduced Snopy, a deep learning method combining change-based sample denoising using vulnerability fixing commits with causal graph learning to enhance vulnerability detection. Snopy employs a Causality Aware Graph Attention Network and Feature Caching Scheme to focus on causal features while suppressing spurious ones. This approach provides a principled way to distinguish causal from non-causal code elements, improving robustness beyond correlation-based detectors. While challenges remain in generalization and spurious correlation mitigation, Snopy represents an important advance toward causal vulnerability detection.

Ganz et al. [10] addressed key practical challenges in ML-driven vulnerability detection by focusing on data quality, model interpretability, robustness, and context sensitivity. They introduced novel neural code augmentation to enhance datasets and leveraged dynamic program analysis to evaluate explanation methods. Their causal learning-based bias assessment framework helped reduce confounding effects and improve detection robustness. This work advances learning-based vulnerability detection toward practical relevance by emphasizing data quality

and interpretability. Related work by Suneja et al. [32] and Yu et al. [43] further explores model interpretability through prediction-preserving input minimization and counterfactual analysis, respectively.

Rahman et al. [29] introduced CausalVul to address the lack of robustness and out-of-distribution generalisation in deep learning based vulnerability detection. They modelled the relationships between code features and vulnerability labels with a causal graph, then used do-calculus and the backdoor criterion in a two-stage pipeline to identify and suppress spurious correlations while preserving causally relevant features. This causal adjustment yields models that rely less on dataset-specific artefacts and more on mechanisms genuinely associated with vulnerabilities, improving both resilience and generalisation. The work represents a clear step toward causal deep learning for vulnerability detection by explicitly targeting false correlations through principled causal inference tools.

Chu et al. [7] 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 actionable insights and guides the necessary modifications to remediate the vulnerability. The primary contribution of the authors was the development of 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. [23].

Islam et al. [13] proposed T5-GCN for vulnerability categorization, localization, and root cause identification by combining graph convolutional networks with a T5-based large language model. Although the work uses the term “root cause,” it does not employ causal deep learning, causal inference, or formal causal reasoning; instead, it relies on DeepLift-SHAP attribution scores to identify tokens that most strongly contribute to the model’s prediction, providing an attribution-based explanation of the vulnerability. The main contribution lies in coupling GCNs with an LLM for code and using explainability techniques to highlight likely root-causing code fragments alongside vulnerability type, location, and a brief static description. This line of work aligns with growing interest in LLM-based code analysis and optimisation [44] and complements efforts such as Pearce et al. [28], who assess the security of code produced by GitHub Copilot, indicating the broader potential of LLMs in vulnerability detection and secure software engineering.

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.

2.1.4 Explainability and Interpretability Techniques

Nguyen et al. [26] introduced GAVulExplainer, one of the earliest efforts to enhance explainability in deep learning-based vulnerability detection. This model-agnostic framework applies to various GNN detectors and uses genetic algorithms to identify subgraphs that most influence a model's prediction by isolating coherent subsets of the Code Property Graph responsible for labeling vulnerabilities. Explanation quality is evaluated with a fidelity metric, and users can adjust explanation size to balance conciseness and completeness. While GAVulExplainer significantly advances interpretable GNN vulnerability detection, it focuses on influential subgraphs rather than explicitly modeling causal relationships between code features and vulnerabilities. Nonetheless, it emphasizes the critical role of explanation fidelity for effective remediation and fostering developer trust.

Li et al. [19] presented Vulanalyzer, a deep learning based framework that targets explainable binary vulnerability detection, a challenging setting due to low level representations and sparse semantic cues. The model combines sequential and topological learning to approximate program execution in assembly, using recurrent units and graph convolutions to preserve instruction semantics and control flow, in line with recent work on functional summaries of assembly code [33]. A multi head attention mechanism highlights influential instructions and basic blocks, guiding developers toward code regions the model considers most indicative of a vulnerability and thereby improving interpretability. However, similar to GAVulExplainer, Vulanalyzer does not incorporate explicit causal reasoning, causal inference, or causal deep learning; attention primarily clarifies *what* features drive predictions rather than *why* those features are causally responsible for the vulnerability [2].

Moschitti et al. [18] presented an XAI based framework for analysing source code in a graph setting, with a focus on how syntactic structures contribute to Common Weakness Enumeration (CWE) classification [17]. Their method ranks

AST constructs by their contribution to specific CWE types for Java and C++ by masking neighbourhoods of code tokens and syntactic nodes in the graph and mapping representation changes to CWE similarity scores via information retrieval techniques. This links learned code feature representations to nuanced semantics recognised by security experts. The authors also identify key limitations of current XAI approaches, including limited generalisation to unseen vulnerability patterns, difficulties in interpreting learned features, and poor transferability across datasets and languages, echoing broader challenges in representing and evaluating large scale code corpora for machine learning on code [1].

Hajipour et al. [16] presented a framework for vulnerability threat prediction that builds semantic representations of vulnerabilities using topic modeling over textual descriptions from sources such as the NVD, and derives an explainable threat score to prioritise analysis. In parallel, they introduced a trend score based on infosec related online discussions to capture emerging, high interest vulnerabilities, aggregating both scores in a visual dashboard to guide security teams toward higher priority cases. While effective for triaging and prioritisation, the approach operates at the description and discussion level, relying on external text and community signals; it does not model the underlying *causal* code level factors that drive exploitability, and may inherit biases and timeliness limitations from its data sources.

Marchetto et al. [24] evaluated how deep learning and explainability methods, in particular SHAP, support vulnerability localization in source code. Using VulDeePecker and JavaBERT as representative DL detectors, they applied SHAP to highlight statements associated with predicted vulnerabilities and found that the resulting explanations often surfaced noisy or only weakly related code features, which could distract rather than assist engineers. This empirical result suggests that current XAI techniques for DL based vulnerability detection frequently expose correlational signals rather than the true underlying causes, underscoring the need for more precise localization mechanisms and the integration of causal reasoning. The authors argue that progress will depend on programming language aware encoding schemes and more advanced techniques specifically tailored to vulnerability localization, rather than generic feature attribution alone.

2.2 Research Challenges and Rationale

Despite substantial progress in deep learning based vulnerability detection, empirical studies consistently report performance degradation under benign refactorings and cross-repository transfer, indicating that many models remain sensitive to spurious, project-specific correlations rather than underlying flaw mechanisms [31, 42]. This limitation highlights a central research challenge: moving beyond correlation centric detectors toward approaches that privilege executable behaviours and causally meaningful flows.

Recent work has begun to introduce causal notions at different levels of the

detection pipeline. Contrastive and counterfactual training and inference schemes, such as *COCA* and *CFExplainer*, use counterfactual variants of code to probe model decisions and reduce reliance on non-robust features [3, 7]. Other approaches, such as *Snopy*, introduce causal adjustment and denoising mechanisms [4]. They filter training samples based on code changes and apply causality-aware graph attention to suppress non-causal context, which reduces spurious correlations in the learned representations. Structural-causal methods, exemplified by *VulCausal* and *CausalVul*, explicitly model relationships between code features and vulnerability labels using causal graphs [15, 29]. These methods then apply backdoor-based corrections to prioritise features with genuine causal influence over dataset artefacts. Collectively, these research works demonstrate the potential of causal inference to improve robustness and generalisation, but they typically operate at the level of feature importance or local predictions rather than reconstructing full executable vulnerability chain across functions and files.

Parallel efforts in explainable AI for software security highlight a gap between saliency-style rationales and truly causal explanations. Studies show that standard attribution and explanation techniques can surface misleading or weakly related code elements, failing to provide developers with trustworthy, causally robust justifications for model predictions [17, 18, 24]. Taken together, these results suggest that many existing explanations reflect correlations in the learned model rather than executable causal mechanisms in the underlying program. This creates two intertwined challenges: evaluating whether explanations are faithful to the model's decision process, and assessing whether they align with executable program behavior under interventions such as code edits, refactoring, and cross-project transfer.

Addressing this gap motivates the chain-centric rationale adopted in this paper. Rather than treating explanations as sets of important features or subgraphs, the proposed approach operates over a unified interprocedural program graph, learns relation-aware evidence propagation on this graph, and then *constrains* decoding so that each explanation is a single executable root→propagation→sink chain. Causal priors and structured decoding help ensure that learned scores translate into behaviors consistent with program semantics and remain robust under counterfactual edits. In this way, the work integrates causal reasoning, interprocedural program representation, and chain-centric explanations to address key challenges in software vulnerability detection.

Chapter 3

Methodology

This chapter describes the approach used in this research to model vulnerability propagation and reconstruct executable root-to-sink vulnerability chains The proposed approach combines causal attention with context-aware graph reasoning over an enhanced CPG that captures both structural and interprocedural relations. It connects root causes to propagation and endpoints via paths that respect program semantics, using pretrained feature initialization, attention-based aggregation on the graph, and causal contextualization to reconstruct accurate, executable root-to-sink chains that are both robust and interpretable.

The pipeline initiates with the construction of an augmented CPG derived from real-world vulnerable repositories. Source code is parsed into ASTs, CFGs, and DFGs, which are subsequently integrated into a unified CPG. This foundational graph is then enriched with interprocedural components, including call-graph edges, argument-to-parameter and return-to-caller mappings, call-site context capturing receiver and dispatch behavior, and coarse alias or points-to summaries, thereby capturing comprehensive data and control flows across procedural boundaries. By progressively enriching the graph in this way, the representation captures how values and control move across function boundaries and modules, so that vulnerability relevant flows can be followed end-to-end rather than being confined to isolated functions. Leveraging the enriched CPG, a structure-aware pretrained encoder (GraphCodeBERT) provides initial token level semantics, which are fused with node level structural features. A relation-aware graph encoder incorporating attention and Adaptive Causal Contextualization (ACC) propagates vulnerability evidence along heterogeneous edges that represent control, data, and call/return relations. This propagation process is guided by a Causal Knowledge Graph (CKG) prior [46], enabling precise modeling of interdependent program semantics. Finally, a constrained beam search decoder explores feasible control- and data-flow paths to assemble executable root-to-sink chains that remain consistent with the interprocedural semantics; the resulting chains and predictions are evaluated using standard detection metrics together with dedicated chain-level validity criteria.

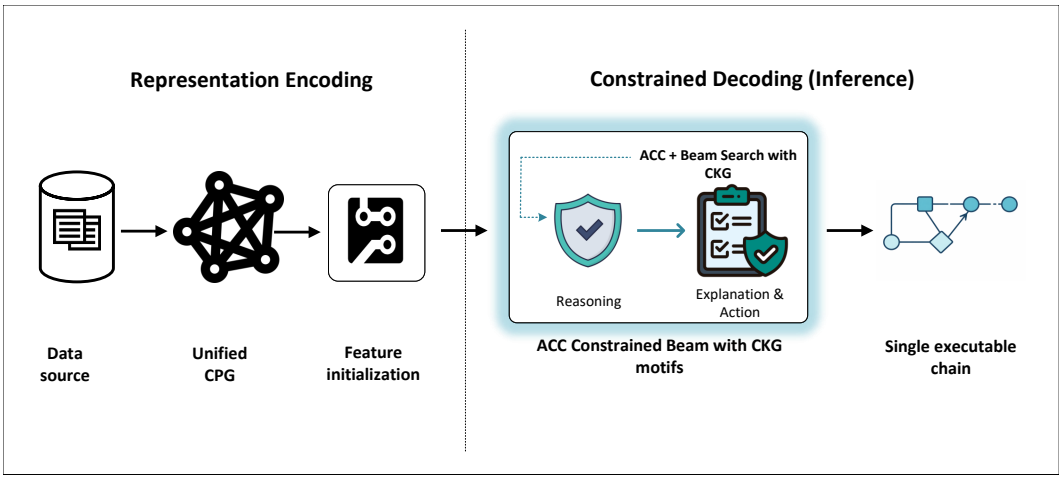

Figure 3.1: Overview of the executable vulnerability chain reconstruction approach.

Figure 3.1 presents a high-level overview of the approach for reconstructing executable root-to-sink vulnerability chain. It emphasizes two core components: representation encoding via an enhanced CPG and constrained decoding for chain reconstruction. The following sections first present the chain-centric program representation, which serves as the foundation for both key components, then detail the model architecture and constrained decoding procedure built upon this representation.

3.1 Chain-Centric Program Representation

A chain-centric, causal approach to vulnerability detection requires a program representation that explicitly models interprocedural mechanisms. Traditional AST, CFG, DFG, and baseline CPG representations enable local reasoning but lack explicit treatment of root-to-sink propagation paths, which limits executable chain reconstruction across functions, files, and modules. This research adopts an augmented CPG that explicitly encodes taint sources, transformations, sanitization, and sinks through control-flow and data-flow, while preserving call/return and aliasing semantics. The representation is designed to be compatible with relation-aware graph encoders and constrained decoding, allowing evidence to propagate along heterogeneous edges and enabling extraction of executable vulnerability chains that support interprocedural causal reasoning.

3.1.1 Dataset and Repository Context: ReposVul

The chain-centric graphs are derived from *ReposVul*, a repository-level benchmark that links vulnerabilities to concrete pre- and post-fix revisions and retains

Table 3.1: Core metadata recorded per ReposVul entry.

Category	Representative fields
Vulnerability entry	CVE-ID, CWE-ID, language, external references, CVE description, publish date, CVSS vector (AV, AC, PR, UI, S, C, I, A)
Patch metadata	Commit ID, commit message and date, project/repository IDs, parent/child links, forge URLs
Related files	File name, language, vulnerable/fixed snapshots, line diffs, file URLs

repository-wide context [36]. Unlike patch-only corpora, *ReposVul* preserves full file snapshots and cross-file dependencies, which is essential for reconstructing long-range interprocedural flows.

Each *ReposVul* entry links CVE/CWE records to precise pre- and post-fix repository snapshots rather than isolated patches [36]. A documented weakness and its severity are anchored to the exact commit, preserving all related files and cross-file dependencies. The corpus is constructed by crawling public sources, disentangling mixed commits to isolate fix-relevant code, and extracting caller–callee relations across translation units. By maintaining chronology and repository-wide call structure, *ReposVul* reveals long-range interprocedural flows essential for chain-centric vulnerability analysis. Table 3.1 summarises the core metadata captured for each entry, including vulnerability descriptors, patch-level information, and file-level snapshots.

3.1.2 Extraction and Graph Construction Pipeline

The conversion from raw dataset entries to chain-centric program graphs occurs over six sequential stages, as illustrated in Figure 3.2.

Stage A: Repository snapshots and vulnerability metadata. CVE/CWE metadata is associated with each *ReposVul* entry, and both parent and child commits are retrieved for every patch. Commit identifiers, messages, dates, repository identifiers, and file paths are indexed so that later stages can reconstruct history-aware cross-file context.

Stage B: Normalization and untangling. Patches frequently mix vulnerability fixes with unrelated refactorings or vendor code. A corpus untangling rule combining model judgements with static cues is applied to retain only files and hunks that are relevant to the vulnerability fix and to discard unrelated changes.

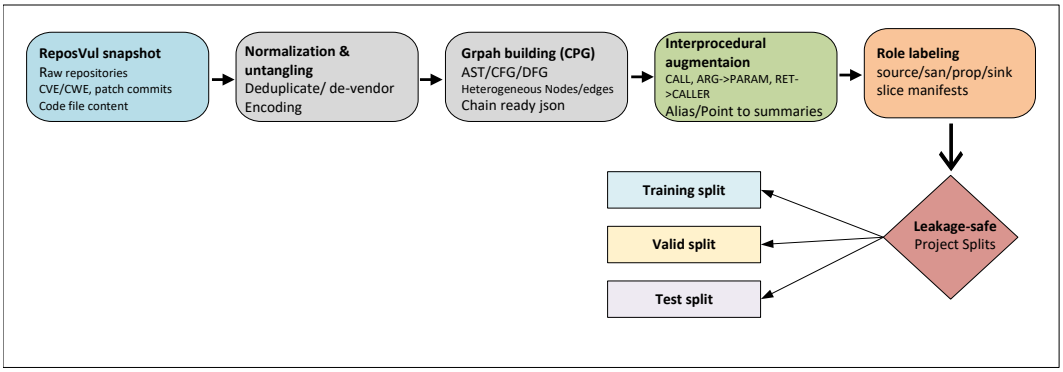

Figure 3.2: ReposVul pipeline: interprocedural CPG construction and leakage-safe data splits.

Stage C: Static graph construction. For each retained snapshot, a heterogeneous program graph is built that overlays AST, CFG, and DFG with direct call edges. Pointer and container def–use are approximated by conservative alias relations, identifiers are anonymized, and literals are bucketed to reduce noise from harmless edits [5, 31]. The result is a chain-ready CPG representation for every project version.

Stage D: Interprocedural augmentation. The CPG is enriched with explicit interprocedural edges, including CALL, ARG→PARAM, and RET→CALLER/RET→LHS, together with summary nodes and alias summaries that connect callee summaries back to call sites. This step lifts per-file graphs to repository-level structures that expose long-range flows from candidate sources to potential sinks.

Stage E: Patch-aware differencing and slice materialization. Line-level diffs are computed for each patch, and four views are synchronised: changed lines, enclosing functions, touched files, and a reachable repository subgraph that can flow to any sink via control or data edges.

Stage F: Leakage-safe filtering and splitting. File paths, repository identifiers, and commit metadata are used to drop broken files, superseded patches, and obsolete snapshots. The remaining instances are grouped into training, validation, and test sets such that leakage across splits is minimised.

Table 3.2 summarises the end-to-end data preparation pipeline, outlining the main stages and their key operations and outputs.

Table 3.2: Preparation pipeline summary.

Stage	Key operations and outputs
A	Link CVE/CWE records to parent and child commits; index commits, projects, and file paths.
B	Normalize and untangle patches; keep only fix-relevant files and hunks; deduplicate.
C	Build a heterogeneous CPG per snapshot by overlaying AST, CFG, DFG, and intra-file call edges; anonymize identifiers and bucket literals.
D	Enrich the CPG with interprocedural links (CALL, ARG→PARAM, RET→CALLER/RET→LHS) and alias summaries to expose repository-level flows.
E	Compute line-level diffs; function, file, and reachable-subgraph slices, with rule-based source/flow/stop/sink labels.
F	Drop broken or superseded patches; form leakage-safe train/validation/test splits at the project level.

3.1.3 Unified Multigraph, Typing, Features, and Storage

Each repository snapshot is represented as a heterogeneous multigraph based on the CPG abstraction [41]. A single typed node store contains program elements (identifiers, literals, statements, basic blocks, functions), and relation-specific edge sets encode AST, CFG, and DFG structure. Interprocedural edges (CALL, ARG2PARAM, RET2CALL, RET2LHS) are materialised in both directions. Graphs are stored in sharded files with repository, commit, and file-path metadata to support scalable provenance and retrieval.

The representation employs a dual-channel feature encoding. A compact structural feature vector captures node type, degree statistics, SSA hints, literal buckets, and role flags, while a 768-dimensional GraphCodeBERT embedding [11] provides contextual token semantics. These two channels are fused during initialization, but the graph topology and interprocedural edge schema remain invariant. The fixed relation alphabet $\mathcal{R}$ used during message passing and decoding includes all intra- and interprocedural relations exploited by the chain decoder (AST, CFG, DFG, CALL, ARG2PARAM, RET2CALL, RET2LHS, and alias summaries).

3.1.4 Ground Truth

Ground truth is defined at two levels. At the revision level, a repository snapshot is labelled vulnerable if it is linked to a corresponding fixed revision in *ReposVul*, which reduces noise from unrelated edits [36]. At the mechanism level, each positive

example carries at least one executable root-to-sink chain that may span multiple functions and files. Sources represent untrusted inputs, sanitizers restrict or reset tainted state, propagators transfer taint through assignments, calls, and returns, and sinks denote security-critical operations. The full role lexicon and matching patterns are listed in Appendix F.

Chains are seeded from patch diffs, extended along def–use links to identify propagators and candidate sinks, and then wired interprocedurally using ARG → PARAM, RET → CALLER, and RET → LHS bindings over the call graph. Retained examples satisfy basic consistency criteria: every source reaches at least one sink, every sanitizer blocks at least one tainted path, and every propagator lies on a CFG-consistent route. A feasibility pass verifies that neutralising the sink reduces exploitability, and a counterfactual pass strengthens guards or replaces dangerous sinks with safe ones to confirm that reachability changes as expected. Cases that fail these checks are corrected or discarded. Split-wise statistics for the C/C++ subset used in experiments are reported in the experimental setup chapter (Section 4.1).

3.1.5 Front-End Analysis Assumptions and Impact

The proposed framework relies on static front-end analyses, including parsing, control-flow graph (CFG) construction, data-flow graph (DFG) extraction, and alias analysis, to build the underlying program representation. As with all static analyses, these components provide conservative approximations of program behavior and may be subject to both over-approximation and under-approximation.

Imprecision in front-end analysis can affect the quality of reconstructed vulnerability chains in several ways. Missing or imprecise data-flow or alias information may prevent certain propagation steps from being recovered, leading to incomplete chains. Conversely, over-approximation may introduce additional feasible paths, resulting in ambiguous or redundant propagation candidates. Such effects primarily influence chain completeness and specificity rather than the overall detection capability.

The framework mitigates these limitations through multiple layers of semantic constraint and validation. Adaptive Causal Contextualization (ACC) enforces consistency with control-flow reachability, data dependencies, call–return discipline, and aliasing constraints during chain construction. Structural validation further filters chains that violate executable semantics, while counterfactual checks assess the causal dependence of the prediction on the reconstructed chain elements. Together, these mechanisms reduce the impact of front-end imprecision by prioritizing semantically coherent and causally supported chains.

The fidelity of the reconstructed chains is intrinsically contingent upon the quality of the underlying static analysis. Consequently, advancements in front-end precision, specifically regarding alias and interprocedural analysis, will directly yield more comprehensive and unambiguous chains, independent of the causal

modeling or decoding framework.

3.1.6 Language Scope and Generality

Although the experimental evaluation in this dissertation focuses on C/C++ programs, the core methodological contributions are not inherently tied to a specific programming language. The chain-centric formulation of vulnerabilities as explicit root–propagation–sink structures, the causality-oriented learning objective, the Adaptive Causal Contextualization (ACC) mechanism, and the constrained decoding and validation procedures operate over semantic program dependencies and are therefore language-agnostic at the modeling level.

Language specificity primarily arises in the front-end analysis stage. Parsing, abstract syntax tree (AST) construction, control-flow graph (CFG) and data-flow graph (DFG) extraction, and alias analysis depend on the syntactic and semantic characteristics of the target language, as well as the availability and precision of analysis tools. In addition, the definition of security-relevant sources, sanitizers, and sinks is language- and library-dependent.

Generalization to other programming languages would require a front-end capable of extracting comparable semantic representations, including control flow, data flow, and interprocedural dependencies. Given such a front-end, the causal modeling, chain construction, decoding, and evaluation components of the proposed framework can be applied without modification. Consequently, while the current instantiation targets C/C++, the approach is flexible in principle and can be extended to other languages subject to the availability of suitable static analysis infrastructure.

3.1.7 Robustness to Code Refactoring

Common code refactoring operations, such as variable renaming, function reordering, modularization, or formatting changes, primarily affect surface-level syntax while preserving the underlying program semantics. Many existing vulnerability detection approaches rely heavily on lexical token sequences or isolated function representations, which makes them sensitive to such syntactic variations and limits their ability to generalize across refactored codebases.

The chain-centric program representation adopted in this paper is designed to mitigate this limitation. Rather than relying on surface tokens, vulnerabilities are modeled through data-flow dependencies, control-flow relationships, and interprocedural call paths captured within the unified multigraph representation. These semantic dependencies remain largely invariant under refactoring operations that do not alter program behavior. As a result, the constructed vulnerability chains preserve their structural and causal integrity across refactored variants of the same codebase, enabling more robust detection in evolving software systems.

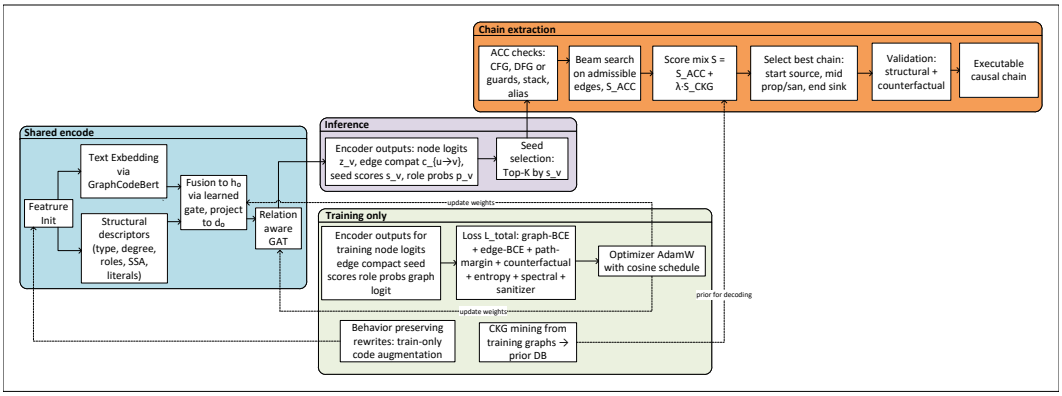

Figure 3.3: Chain-Centric Model Architecture and Inference Flow

3.2 Model Architecture and Decoding Pipeline

This section explains how source code and program structure are fused, encoded, and decoded into a single executable chain. All equations and symbol glossaries are consolidated in Appendix B. Refer to Figure 3.3 for a comprehensive single-page overview that this section follows.

Node features are initialized with code spans and structural descriptors, encoded via a relation-aware Graph Attention Network (GAT), and decoded into a single executable chain subject to ACC feasibility. Inference begins with seeds selected by the model and can incorporate a Causal Knowledge Graph (CKG) prior for ranking. Training optimizes a composite loss, using CKG derived only from the training split. Solid arrows indicate data or score flow, dashed arrows denote training-only operations, and dotted arrows represent weak decoding priors or optimizer feedback to trainable encoder components. GraphCodeBERT provides pretrained contextual embeddings for code tokens, which are fused with structural descriptors before graph encoding.

3.2.1 GraphCodeBERT Feature Initialization

Each node aligned to a concrete source span is tokenized with GraphCodeBERT's byte-pair encoding and encoded in sliding windows (length L, stride S). Window-level contextual token embeddings are aggregated back to the node: if the span contains identifiers, a light edge-aware attention emphasizes tokens carrying def–use signal (Appendix Eqs. B.2–B.3); otherwise a simple average is used (Appendix Eq. B.1). Spans that appear in multiple windows are pooled per window and then averaged, and function-proxy nodes pool their child statements to obtain a coarse function representation.

The textual embedding $\mathbf{x}_i^{\text{text}}$ and structural descriptor $\mathbf{x}_i^{\text{struct}}$ (node type, degrees,

Table 3.3: Node feature dimensions at initialization.

Component	Dim.	Notes
Textual embedding	768	GraphCodeBERT contextual vector [11]
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 LN)
Projected structural	d_0	$W_s : \mathbb{R}^{d_s} \to \mathbb{R}^{d_0}$ (with LN)
Fused node init $h^{(0)}$	d_0	Gated combination for the GAT encoder

SSA hints, literal statistics, role flags) are both projected to dimension d_0 and fused via a learned gating mechanism to obtain the initial node state $\mathbf{h}_i^{(0)}$ (Appendix Eqs. B.4–B.6). When a node and its def–use neighbours fall within a single window, GraphCodeBERT's data-flow mask is enabled so these pairs can attend directly; the encoder remains frozen to attribute performance gains to the graph module, and Table 3.3 summarizes the resulting feature dimensionalities.

3.2.2 GAT with Causality-Oriented Attention

A relation-aware Graph Attention Network consumes the fused initializations and builds contextual node states while learning how different edge types contribute to aggregation. Per layer, attention and updates follow Appendix Eqs. B.7–B.8. After L layers the encoder emits a node vulnerability logit z_v, a seed score s_v, and relation-gated edge compatibilities $c_{u\to v}^{(r)}$ (Appendix Eqs. B.9–B.10). Seeds identify likely chain starts as the top-K nodes by s_v (Appendix Eq. B.11); candidate paths are scored by a log-additive mixture of node evidence and edge compatibility (Appendix Eq. B.12), with α controlling the mix. Training couples node- and edge-level objectives with a path-margin term so that true chains outrank admissible distractor walks.

3.2.3 Causal Knowledge Graph (CKG): Mining and Prior

From training chains only, unigram and bigram statistics are mined over the relation alphabet. At decoding time, admissibility is unchanged, but the ranking of admissible expansions receives a small prior bonus $S_{\mathrm{CKG}}(\pi)$ mixed into the path score with weight λ (Appendix Eq. B.13). This preserves data-driven evidence while gently preferring historically plausible relation patterns.

3.2.4 Adaptive Causal Contextualization (ACC)

ACC converts encoder scores into a single executable interprocedural chain by enforcing constant-time feasibility checks and role-shaped preferences. A partial path maintains a taint footprint, a call stack, and accumulated guards; an expansion

$u \xrightarrow{r} v$ is admissible only if all predicates hold (Appendix Eqs. B.14–B.18). The path score is adjusted by start/middle/end role penalties and a sanitizer dominance bonus (Appendix Eqs. B.19–B.23), together with mild length and repetition costs (Appendix Eqs. B.24–B.25), yielding the ACC objective $S_{\mathrm{ACC}}(\pi)$ and its optional CKG mixture $S^{\star}_{\mathrm{ACC}}(\pi)$ (Appendix Eqs. B.26–B.27). On `CALL`, the callee and site are pushed; `RET2CALL`/`RET2LHS` pop only with matching sites, preserving well-nested cross-function paths. The decoding complexity with beam width B, horizon H, and average admissible out-degree $\bar{d}$ scales as $O(B \cdot H \cdot \bar{d})$. This reflects exploring B candidates over H steps, each with $\bar{d}$ possible expansions, consistent with standard beam search computational cost analyses.

3.2.5 Chain Extraction and Validation

A candidate is valid if it starts near a source, contains at least one interior propagator or sanitizer, and ends at a sink (Appendix Eqs. B.28–B.30); when interprocedural links exist in the slice, at least one must appear in the chain. Among admissible paths across beams from the top-K seeds, selection maximizes $S_{\mathrm{ACC}}(\pi)$ (Appendix Eq. B.31). The final chain is then validated structurally and counterfactually.

3.2.6 Interpretability and Explanation Generation

The proposed work provides interpretability by explicitly reconstructing executable, interprocedural chains that explain how a vulnerability arises and propagates through the program. Rather than relying on post-hoc token attribution or isolated importance scores, explanations are generated as structured causal paths that connect a root source to a security-sensitive sink via concrete program statements and dependencies.

Interpretability emerges from three tightly coupled components. First, the causality-oriented attention mechanism assigns importance to nodes and relations based on their contribution to vulnerability propagation under the ACC objective. These scores bias both node selection and path expansion during decoding, ensuring that high-attention elements are preferentially included in candidate chains. Second, Adaptive Causal Contextualization refines these scores using control-flow, data-flow, and interprocedural constraints, suppressing spurious correlations that do not correspond to executable behavior. Third, ACC-constrained beam decoding produces a linearized chain that respects call–return discipline, data dependencies, and guard conditions.

The resulting explanation is therefore not an abstract score but a concrete artifact: an ordered sequence of statements linked by data-flow, control-flow, and interprocedural edges. Each step in the chain corresponds to a specific source location and semantic role (e.g., source, propagator, sanitizer, or sink), and the entire chain is validated both structurally and counterfactually as described in Section 3.2.5. This representation enables developers to directly trace how untrusted input reaches

a sensitive operation and to identify the precise program locations responsible for the vulnerability.

3.2.7 Scope and Limits of Causal Guarantees

The causal framing adopted in this research is intended to capture *causal faithfulness* with respect to program semantics rather than to provide absolute guarantees of concrete runtime execution. Specifically, the reconstructed root–propagation–sink chains are faithful to the static control-flow, data-flow, and interprocedural dependencies extracted from the program and validated under the structural and counterfactual constraints described earlier in this chapter.

It is important to distinguish this notion of causal faithfulness from true runtime causality. The generated chains do not assert that a particular execution will occur for all inputs, environments, or schedules, nor do they replace dynamic analysis or execution tracing. Instead, they represent semantically consistent vulnerability propagation paths that are executable under the modeled program semantics and constraints.

This intermediate position between purely correlation-driven learning approaches and fully dynamic analysis is deliberate. By grounding predictions in executable semantic dependencies while remaining scalable to large codebases, the proposed approach avoids spurious explanations and supports actionable reasoning about vulnerability mechanisms, without over-claiming runtime guarantees beyond what static analysis and learned models can provide.

3.2.8 Training, Optimization, and Hyperparameters

The total loss combines graph-level binary cross-entropy with regularizers for flow and counterfactual consistency, attention entropy, spectral control, and sanitizer alignment. The CKG prior is applied only at decoding time. Models are trained with AdamW, cosine learning-rate decay, gradient clipping, mixed precision, and weighted sampling for class imbalance. Semantics-preserving augmentations (renaming, benign reordering) are used, and fixed seeds, logged checkpoints, cached features, and archived training curves support reproducibility.

This chapter summarizes a five-step methodology for reconstructing executable root-to-sink vulnerability chains. First, nodes are encoded with GraphCodeBERT and fused with structural features to obtain context-sensitive initial states for all program elements (Appendix Eqs. B.2–B.6). Second, a relation-aware graph attention network propagates information along heterogeneous control-, data-, and call/return edges so that node representations reflect their interprocedural context (Appendix Eqs. B.7–B.8). Third, a constrained decoder guided by a weak causal knowledge graph prior scores and assembles candidate paths, restricting search to syntactically and semantically plausible root-to-sink sequences (Appendix Eq. B.13). Fourth, adaptive causal contextualization (ACC) filters and selects chains that satisfy

control-flow, data-flow, and aliasing constraints, ensuring that only semantically valid mechanisms are retained (Appendix Eqs. B.14–B.31). Finally, the entire model is trained at the graph level using binary cross-entropy loss with light regularization (Appendix Eq. B.32), aligning vulnerability predictions with the reconstructed executable chains.

Chapter 4

Experimental Results and Analysis

This chapter presents the experimental results for the proposed chain centric interprocedural vulnerability detector. Under the default `ReposVul` splits, it quantifies graph level detection performance, the quality of reconstructed root–propagation–sink chains, causal consistency under targeted interventions, and inference cost and decoding behaviour. All results are obtained under a single, leakage-safe experimental setup and are averaged over five seeds to support reproducibility.

The analysis progresses from a global behavioral overview to a detailed chain level examination. It begins by summarizing the dataset, model variants, and evaluation metrics used throughout the experiments, then reports detection metrics for both encoder configurations. Subsequent sections investigate executable chain feasibility and interprocedural structure, assess causal faithfulness via counterfactual edits, and evaluate decoding efficiency. The chapter concludes with robustness experiments and an example illustrating the reconstruction of a root–propagation–sink vulnerability chain.

4.1 Experimental Setup and Configuration

This section summarizes the experimental setup used throughout the evaluation, including datasets and default splits.

Dataset and splits: All experiments use the default splits provided with `ReposVul` [36]. Splits are repository-disjoint, so code from the same repository does not appear in multiple splits. Positive and negative pairs are kept within the same split, and, when reported, chronology is respected so that training data does not depend on future revisions.

Tables 4.1, 4.2, and 4.3 report split-wise statistics for the prepared C and C++ subset used in the experiments, including label balance at the file-snapshot level, graph and node label density after chain-centric conversion, and the prevalence of

Table 4.1: File-level label counts by split (C/C++ subset).

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 4.2: Graph instances and node-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}

interprocedural caller/callee structure.
These statistics indicate that roughly one third of instances have at least one callee, about one eighth have at least one caller, and around one tenth have both, providing the minimal interprocedural structure required to reconstruct vulnerability chains that span function boundaries.

Inputs and graph relations: Models operate on chain-prepared interprocedural CPGs that overlay ASTs, CFGs, and DFGs, and include interprocedural relations `CALL`, `ARG→PARAM`, `RET→CALLER/RET→LHS`, together with alias summaries. Table 4.4 summarizes the main graph, encoder, decoding, training, and calibration settings shared across all experiments.

Model variants: Two encoder configurations are compared under the same graph topology and training/decoding regime. The *Struct-only* variant uses compact structural features (types, degrees, SSA hints, literal buckets), while *GCBERT+Struct* augments these with frozen GraphCodeBERT embeddings [11]. Table 4.5 contrasts the resulting graph topology and feature dimensionality on a representative shard.

Table 4.3: Interprocedural connectivity (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

Table 4.4: Experimental Setup Details.

Item	**Setting**
Dataset	`ReposVul`, default repository-disjoint splits, chronology respected
Graph relations	DFG, CFG, CALL, ARG2PARAM, RET2CALL, RET2LHS; alias summaries enabled
Encoder	Relation-aware GAT, width d_0=64, L=3 layers
Variants	Struct-only; GCBERT+Struct (GraphCodeBERT frozen)
Decoding	Beam with ACC, K=8, B=24, H=5, node/edge mix α=0.7
CKG prior	Decoding-only mixture λ=0.2; smoothing 10^{-3}; temperature 1.0
Training	AdamW; LR 2×10^{-3}; WD 10^{-4}; early stop on validation macro-F1
Calibration	Temperature scaling on validation; ECE with 15 bins
Thresholds	$\tau_{\mathrm{F1}^\star}$ (per variant) and τ=0.5
Seeds	5 per configuration; mean and 95% CIs reported
Environment	PyTorch 2.4.1, CUDA 12.1, RTX 4070 Laptop GPU, AMP on
Embeddings cache	GraphCodeBERT features, FP16, max length 512, stride 384

Table 4.5: Comparison of Struct-only vs. GCBERT+Struct encodings.

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

Table 4.6: Decoding and Training Configuration.

Item	Setting
Beam / horizon / mix	K=8, B=24, H=5, node–edge mix α=0.7
Admissibility gates	CFG reachability, ARG→PARAM, RET→CALLER/LHS, alias checks, stack discipline
CKG prior (inference only)	Mixture λ=0.2; smoothing ϵ=10^{-3}; temperature τ=1.0; top-K trigrams = 500; weights $(\beta_1, \beta_2, \beta_3)$=$(0.3, 0.6, 0.1)$
Loss	Class-weighted BCE at graph level + flow and causal regularizers
Early stopping	Macro-F1 on validation, patience = 5
Encoder freeze	GraphCodeBERT frozen; ablation unfreezes last 2 blocks at 0.1× LR
Thresholds	$\tau_{\mathrm{F1}^\star}$ from validation and fixed τ=0.5 on test
Calibration	Temperature scaling fit on validation; ECE with 15 bins
Seeds & uncertainty	5 seeds; mean and 95% CIs; bootstrap 10^4 for metrics, binomial CIs for DCR, paired bootstrap with Cliff's δ

Decoding and evaluation: Decoding uses ACC-constrained beam search, allowing only moves that satisfy control-flow reachability, consistent ARG→PARAM and RET→CALLER/LHS bindings, alias checks, and stack discipline. A causal knowledge graph (CKG) prior mined from training chains adjusts move scores during inference but does not alter admissibility. Training uses class-weighted binary cross-entropy at the graph level with flow and causal regularizers, early stopping on validation macro-F1, and calibrated thresholds derived from validation data. Table 4.6 summarizes the decoding, training, and evaluation configuration.

Experiment Environment and Artifact: Experiments were run on an NVIDIA RTX 4070 Laptop GPU with CUDA 12.1 and PyTorch 2.4.1 using automatic mixed precision. GraphCodeBERT embeddings were precomputed and cached as FP16 tensors. The artifact archive includes configuration files, predictions, calibration parameters, causal intervention logs, beam expansion analyses, and environment metadata. This setup ensures full reproducibility of the results, with all decoding, calibration, and reporting details preserved.

4.2 Evaluation Metrics

This section defines the standard classification, chain-centric, and counterfactual metrics used to assess detection performance, executable chain quality, and causal faithfulness of the proposed approach.

4.2.1 Standard Classification Metrics

Classification models are commonly evaluated by thresholding predicted probabilities to produce discrete labels, enabling the construction of a confusion matrix from counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). From these counts, primary metrics such as Precision, Recall, and F1 score are computed:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{F1} = \frac{2\,\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{4.1}$$

Precision quantifies the accuracy of positive predictions, Recall measures the coverage of actual positives, and F1 balances the two in a single harmonic mean. Additionally, metrics including Accuracy, Macro-/Micro-F1, AUROC, and AUPRC are computed on the raw prediction scores without thresholding. Due to the low prevalence of vulnerable instances, AUPRC and Macro-F1 are emphasized as primary indicators of detection performance. Formal definitions are detailed in Appendix C.2.

4.2.2 Chain-Centric Metrics

For each positive decision, at most one executable chain is returned (or none if feasibility is not met).

Validity: The fraction of returned chains that satisfy ACC feasibility checks is reported. These checks include CFG reachability, def–use or guard consistency, interprocedural call–return discipline, and alias coherence. The formal definition is in Appendix. C.3.

Structural fidelity: When a reference chain is available, reported metrics include node and edge coverage, longest common subsequence ratio (LCS), and role-aware coverage for source, sanitizer, propagator, sink. Formulas are detailed in Appendix C.3.

Interprocedurality: The IPA Rate quantifies the share of predicted chains that traverse `CALL/ARG`→`PARAM/RET` edges when such edges exist in the slice (Appendix. C.3).

4.2.3 Counterfactual Metrics

Causal robustness is evaluated through three targeted interventions: guard strengthening, call unbinding, and sink neutralization. Results are reported by intervention type with confidence intervals; detailed methods are provided in Appendix C.4. The CKG prior is active except during robustness tests, when it is disabled on the edited graph.

Counterfactual Consistency Score (CCS):

$$\mathrm{CCS}_i = (p_i - p_i^{\mathrm{do}})^2, \tag{4.2}$$

where p_i and p_i^{do} denote the predicted probabilities before and after the targeted intervention on graph i, respectively, both bounded in $[0, 1]$. A low CCS indicates minimal change in prediction, while a high CCS signals a substantial effect of the intervention. Appendix C.4 further defines a directional consistency rate assessing whether changes occur in the expected direction per intervention type.

Causal Feature Attribution Measure (CFAM):

$$\mathrm{CFAM}_i = \frac{\sum_{f \in F_c} A_i(f)}{\sum_{f \in F_c \cup F_s} A_i(f)} \in [0, 1], \tag{4.3}$$

where F_c and F_s are sets of on-chain and off-chain features, respectively, and $A_i(f) \geq 0$ denotes the attribution score of feature f in graph i. CFAM quantifies the fraction of total attribution attributed to features on the returned causal chain. Values near 1 indicate strong alignment of attribution with the chain's features. Normalization details are given in Appendix C.4. Auxiliary decoding diagnostics, including Chain Success Rate, admissible expansion ratio, and prior influence rate, are defined in Appendix C.5.

4.3 Detection Results

For both encoder variants, this section reports standard graph-level metrics: Accuracy, Precision, Recall, F1, AUROC, and AUPRC. The operating threshold $\tau_{\mathrm{F1}^\star}$ is selected on the validation split and applied unchanged to the test set. Probabilities are calibrated via temperature scaling on validation and reused on test, and calibration quality is summarised using Expected Calibration Error (ECE), where lower values indicate better alignment between predicted confidence and empirical accuracy (formal definition and binning scheme in Appendix C.1). AUPRC is treated as the primary metric, with F1 and AUROC providing additional context given the class imbalance in the test set (positives $\approx 2.9\%$). All results are averaged over five random seeds.

Table 4.7 reports validation and test performance at $\tau_{\mathrm{F1}^\star}$ (mean over five seeds) for both encoder variants, including thresholded Accuracy, Precision, Recall, F1, and the corresponding AUROC and AUPRC values.

Across both valid and test splits, the GCBERT+Struct configuration yields higher F1 and AUPRC than Struct-only (AUPRC +0.15, F1 +0.14 on Valid; AUPRC +0.15, F1 +0.13 on Test). Accuracy remains high for both variants due to class imbalance, while AUROC indicates stable ranking performance.

Table 4.7: Valid and test metrics for Struct-only and GCBERT+Struct.

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

Table 4.8: Operating thresholds and calibration parameters.

Split	Variant	$\tau_{F1^\star}$	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	1.41	0.082	0.036
Test	GCBERT+Struct	0.27	1.29	0.064	0.022

Table 4.8 summarises the corresponding operating thresholds and calibration parameters, including the learned temperature T and ECE values before and after temperature scaling.

Calibration reduces ECE for both variants, and the relative ordering between Struct-only and GCBERT+Struct is consistent with the improvements observed in AUPRC and F1. Precision–Recall and ROC curves exhibit the same pattern. To contextualise these aggregate metrics, Table 4.9 presents confusion matrices on TEST at $\tau_{F1^\star}$ for both encoder variants.

False positives correspond to safe code incorrectly flagged as vulnerable, while false negatives indicate missed vulnerabilities. In this setting, many false negatives arise in macro-expanded code or callback-driven execution paths, where control flow is implicit and interprocedural structure is more difficult to recover faithfully.

Table 4.9: Confusion matrices on TEST at $\tau_{F1^\star}$ for Struct-only and GCBERT+Struct.

	Struct-only		GCBERT+Struct	
	Pred. Neg	Pred. Pos	Pred. Neg	Pred. Pos
True Neg	21,779	775	22,008	546
True Pos	322	348	241	429

Table 4.10: Feasibility of reconstructed chains (pass of all 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

Table 4.11: Interprocedural structure in predicted chains.

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

4.4 Executable Chain Quality and Interprocedural Evidence

This section evaluates the quality of reconstructed chains using feasibility, interprocedural usage, structural agreement with ground truth, and chain length/span statistics. All quantities are computed on positive slices and averaged over five seeds. Table 4.10 reports the fraction of predicted chains that pass all ACC feasibility checks (control-flow reachability, call/return consistency, data/guard preservation, alias coherence).
GCBERT+Struct improves chain feasibility on both validation and test splits. Table 4.11 summarizes interprocedural usage: the IPA rate (share of chains that use any CALL/ARG→PARAM/RET edges), the fraction that include both call and return edges, and the mean call depth.
The enriched encoder yields higher IPA rates, more chains that use both call and return edges, and slightly deeper call stacks.

Table 4.12 reports node and edge coverage, together with role-aware coverage for *source*, *sanitizer*, *propagator*, and *sink* nodes against the reference chains, and Table 4.13 summarises order agreement using the longest common subsequence (LCS) ratio.
GCBERT+Struct improves both coverage (especially for sanitizers and propagators) and order agreement. Finally, Table 4.14 characterises chain length and span via hop count, number of files crossed, and the proportion of summary edges.
Chains remain short and typically span one to two files, with similar hop counts across variants. The enriched encoder slightly reduces reliance on summary edges while increasing feasibility, interprocedural usage, and agreement with ground-truth chains.

Table 4.12: Role and edge agreement with ground truth.

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 4.13: Order agreement via LCS ratio.

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

Table 4.14: Chain length and span: hops, files crossed, and summary-edge share.

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

Table 4.15: Prototype causal metrics (means over 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	Default τ=0.25
Test	7.97×10^{-9}	0.0233	Same thresholding

Table 4.16: CCS by edit type (lower is better for minor, higher for sink edits).

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

4.5 Causal Faithfulness

This section evaluates how strongly predictions depend on the reconstructed chains using counterfactual interventions. The *Counterfactual Consistency Score* (CCS) measures the magnitude of probability changes under edits, the *Directional Consistency Rate* (DCR) captures how often changes move in the expected direction, the *Causal Feature Attribution Measure* (CFAM) quantifies how much attribution mass lies on the predicted chain, and the chain invalidation rate records how often no valid chain remains after editing. Formal definitions and variants are given in Appendix C.4 and.

Table 4.15 reports prototype CCS and CFAM values averaged over a small subset of graphs, illustrating the scale of probability shifts and on-chain attribution at different training stages.

Table 4.16 decomposes CCS by intervention type: strengthening guards and unbinding calls (minor edits) versus sink neutralisation (major edit). Minor edits produce small probability shifts, while sink edits cause substantially larger changes, indicating that the endpoint of the chain has strong influence on the decision.

Table 4.17 shows the Directional Consistency Rate (DCR), that is, the proportion of edits where the probability change has the expected sign. DCR is high for all edits, particularly for sink interventions, indicating stable qualitative responses to causal perturbations.

Table 4.18 presents CFAM scores overall and decomposed by chain segment (source, sanitizer, propagator, sink), with values in $[0, 1]$. Higher values indicate that a larger share of attribution mass lies on nodes that belong to the returned chain.

Table 4.19 presents chain invalidation rates, defined as the fraction of instances where no admissible causal chain remains after applying an edit. It also reports the average change in chain score following sink neutralization interventions. The

Table 4.17: 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

Table 4.18: CFAM (overall and by chain segment). 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

results show that minor edits invalidate approximately half of the chains, indicating moderate disruption of causal explanations. In contrast, sink neutralization almost completely invalidates the chains and results in substantial decreases in chain scores, reflecting the significant impact of this intervention on causal model structure and confidence. This highlights the differing robustness of causal chains to various types of graph edits and underscores sink neutralization as a more drastic perturbation within causal robustness evaluation.

Taken together, these results demonstrate that the model's decisions are strongly linked to the reconstructed causal chains. Minor edits along the chains induce small, directionally consistent shifts in predicted probabilities, reflecting stable and interpretable causal influence. In contrast, major edits that neutralize the sink disrupt this causal structure, invalidating most chains and producing substantial probability decreases. This indicates the model relies heavily on these causal chains for its decision-making, with robustness and sensitivity aligned to the nature and location of the intervention within the chain.

Table 4.19: Chain invalidation rates 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

Table 4.20: Inference latency per graph.

Component	Struct-only	GCBERT+Struct	Notes
Encoder forward (ms)	18.6 (31.9)	31.7 (52.5)	Relation-aware GAT; LM frozen
ACC decoding (ms)	2.8 (4.6)	3.3 (5.3)	Admissibility gates
CKG prior mix (ms)	0.3 (0.5)	0.4 (0.6)	Lightweight lookups
Total (ms)	**21.8** (36.8)	**35.6** (58.4)	End-to-end latency

4.6 Efficiency and Decoding Dynamics

This section reports the computational cost of the proposed pipeline and the effect of decoding constraints on search behaviour. Table 4.20 summarises per-graph inference latency for the main components. Most of the runtime is spent in the encoder forward pass, while ACC checks and the CKG prior incur only a small additional overhead.

Structural and ACC checks prune most candidate moves during decoding: CFG reachability alone filters out around 54–63% of expansions (Appendix D.2, Table D.2). Increasing the beam from $(4, 12, 3)$ to $(8, 24, 5)$ improves chain validity, interprocedural usage, and LCS (by 4.2, 13.1, and 6.3 points respectively) at an additional 6.9 ms per graph. Given that encoder computation dominates latency, the chosen beam setting provides a practical trade-off between chain quality and runtime.

4.7 Robustness and Generalization

Robustness is assessed under three conditions: cross-repository generalization, forward-in-time evaluation, and invariance to benign refactorings, using the default `ReposVul` splits and five random seeds. Across unseen projects and later revisions, graph-level detection metrics change only slightly, while chain validity and interprocedural usage remain high and track classification trends within overlapping confidence intervals. Under refactorings (identifier renaming, inert code insertion, and mild intra-block reorderings), predictions and chains are largely stable; only aggressive statement reordering degrades performance noticeably. Detailed numerical deltas and confidence intervals are reported in Appendix D.

4.8 Test Case

This case study illustrates how the model behaves in practice by examining a decoded interprocedural chain from root to sink in a two-file program composed of `main.c` and `lib/shell.c`. Figure 4.1 visualizes the decoded interprocedural chain from root to sink discussed in this case study. The example uses the same trained model and

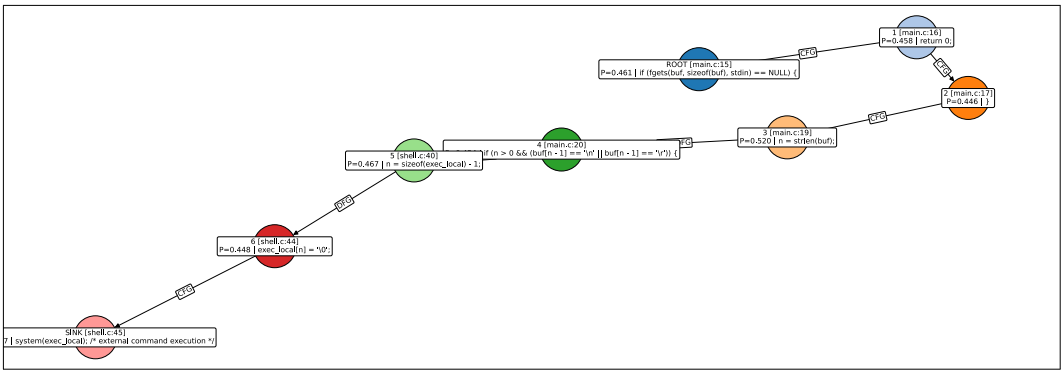

Figure 4.1: Executable interprocedural chain from root to sink across `main.c` and `lib/shell.c`

ACC-constrained decoding with the default beam settings as in the main experiments. It demonstrates the interpretability mechanism described in Section 3.2.6. The full textual trace and exact source files are provided in Appendix G.10.

In this example, untrusted input enters the program via the call `fgets(buf, sizeof(buf), stdin)` in `main.c`. This call is the *root* of the chain, introducing external data into the program state. The chain then follows local processing of `buf`: computing `n = strlen(buf)` and conditionally trimming trailing newline or carriage return characters. These steps provide lightweight sanitization and normalization, but the data in `buf` remains marked as tainted, meaning it is considered untrusted and potentially harmful throughout the program's execution.

The cleaned string is then passed to the execution stage, with the chain crossing the file boundary via a call to `stage_execute` and continuing in `lib/shell.c`. ACC permits this transition only if a valid call edge exists from `main.c` to `lib/shell.c`. Within `stage_execute`, the chain includes calculating the copy length `n = sizeof(exec_local) - 1`, writing the null terminator `exec_local[n] = '\0'`, and preparing the fixed-size buffer `exec_local`. These steps show how the tainted command string is copied into a local buffer that will be executed later.

Finally, the chain terminates at the call `system(exec_local)`, which is labeled as the *sink*. This is a security-sensitive operation, since it executes a shell command derived from external input. Every step along the chain corresponds to a concrete statement in the source code, and edges represent either control flow, data flow, or interprocedural links (calls and returns). ACC enforces that each hop is reachable on the control-flow graph, respects the data and guard dependencies, and maintains call/return discipline and alias constraints. As a result, the final path forms an executable, interprocedural explanation of how attacker-controlled input flows from `fgets` in `main.c` all the way to `system(exec_local)` in `lib/shell.c`.

4.8.1 Limitations and Ambiguous Reconstructions

While the reconstructed causal chains are executable and semantically coherent, certain cases exhibit incomplete or ambiguous reconstructions. These situations typically arise from conservative approximations in static front-end analysis, such as imprecise alias information, over-approximate call targets, or partial data-flow recovery. In such cases, the decoder may produce shorter chains that terminate before reaching the true sink or multiple competing chains that satisfy semantic constraints with comparable confidence.

Ambiguity may also occur in the presence of partial or context-dependent sanitization, where data is conditionally cleaned along some paths but remains tainted along others. The resulting chains reflect this uncertainty by including propagation steps that are semantically valid but not uniquely decisive. Importantly, these outcomes do not indicate spurious correlations; rather, they expose genuine uncertainty inherent in static program analysis and interprocedural reasoning.

Despite these limitations, the framework continues to provide meaningful diagnostic value. Even when chains are incomplete or ambiguous, they localize high-risk propagation regions and preserve interpretability through explicit semantic structure. These cases are particularly useful for inspection dynamic analysis, reinforcing the approach as a transparent and practical aid for vulnerability analysis rather than a replacement for full runtime verification.

The results in this chapter demonstrate competitive detection performance with strong AUPRC and F1 scores. Constructed chains show high feasibility and consistency. Counterfactual analysis confirms causal fidelity. Inference is practical, dominated by encoder latency with minimal decoding overhead. Performance remains stable under project-disjoint splits and rigorous conditions. Complete diagnostics and reproduction artifacts are provided in Appendix G.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This dissertation systematically investigated a chain-centric, causality-driven approach for interprocedural software vulnerability detection. The central objective was to demonstrate that vulnerabilities can be modeled as explicit root–propagation–sink causal mechanisms reconstructed directly from real-world source code, enabling predictions that are reliable, interpretable, and grounded in program semantics rather than superficial correlations.

The core contribution of this work lies in conceptualizing vulnerability detection as the identification and validation of executable causal chains. By representing programs using an interprocedural, augmented Code Property Graph and framing vulnerabilities as structured propagation paths, this dissertation moves beyond conventional token- or function-centric classification paradigms. The proposed work integrates causal reasoning into both representation and inference, allowing predictions to be accompanied by concrete, semantically faithful explanations that reflect how vulnerabilities arise and propagate across procedural boundaries.

Empirical evaluation demonstrates that this causal, chain-centric formulation is effective in practice. Across repository-disjoint settings, the approach achieves competitive detection performance while consistently reconstructing executable and interprocedural chains that align with expected vulnerability semantics. Targeted causal diagnostics and counterfactual analyses further confirm that model decisions are driven by the reconstructed propagation mechanisms rather than spurious contextual cues, supporting the claim of causal faithfulness. Importantly, these gains are achieved without imposing prohibitive computational overhead, indicating that the framework is suitable for practical use.

Taken together, this work establishes a unified methodological foundation for interprocedural vulnerability detection that tightly couples representation, learning, and evaluation through causal reasoning. By shifting the focus from opaque classification toward explicit causal explanation, the proposed approach advances

both the theoretical understanding of software vulnerability propagation and the practical design of robust, interpretable, and actionable security analysis tools for modern software systems.

5.2 Future Work

Although this work advances causality-aware vulnerability detection, several extensions remain. A natural next step is to move beyond single-chain explanations and support multiple distinct causal chains per vulnerability instance. Real-world vulnerabilities often permit several exploit paths, and producing a small set of diverse, non-overlapping chains would give analysts a more complete picture of risk without introducing unnecessary complexity.

Another key direction is improving the priors and semantics that guide decoding. The current CKG is a fixed prior, and making it learnable or jointly trainable could better capture project-specific patterns. Likewise, refining the interprocedural representation with more precise aliasing, improved modeling of dynamic dispatch, and framework-aware summaries would yield chains that more closely reflect actual runtime behavior and increase their practical utility.

Incorporating lightweight dynamic evidence presents another promising avenue. Static analyses may identify vulnerability propagation paths that are feasible in theory but unlikely to occur in realistic executions. By integrating runtime information the system could prioritize and filter generated chains, assign probabilistic scores to chain segments, and verify the executability of reported paths under realistic conditions. This integration would bridge the gap between static interprocedural analyses and observed program behavior, thereby increasing confidence in the detection results.

Expanding the current approach beyond its primary focus on C/C++ code is essential to assess its broader applicability. Extending to additional programming languages, paradigms, and vulnerability classes including concurrency defects, logic errors, and vulnerabilities arising in multi language stacks would probe the generality of the chain-centric causal reasoning philosophy. Concurrently, integrating the approach into developer toolchains and conducting empirical user studies could provide insight into its practical impact, including triage efficiency, fix accuracy, and developer trust in automated vulnerability explanations.

Evaluation methodology remains a critical frontier. While causal quality and faithfulness criteria establish a foundation for chain-level validation, future work should refine these metrics, benchmark across diverse model families, and ground them in human-centered outcomes. Establishing community-adopted benchmarks with explicit chain annotations, standardized splits, and unified evaluation criteria will enable rigorous comparison and advance vulnerability detection systems that balance predictive accuracy with interpretability.

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 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), 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 - 2024 39th ACM/IEEE International Conference on Automated Software Engineering, ASE 2024, 2024, pp. 606–618.

[5] 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.

[6] 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.

[7] 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.

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

[9] A. H. Galib and B. M. Mainul Hossain, *A systematic review on hybrid analysis using machine learning for android malware detection*, in 2019 2nd International Conference on Innovation in Engineering and Technology (ICIET), 2019, pp. 1–6.

[10] 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.

[11] 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).

[12] 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.

[13] 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).

[14] A. Kaur and R. Nayyar, *A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code*, Procedia Computer Science, 171 (2020), pp. 2023–2029.

[15] 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.

[16] 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.

[17] D. Li, *An XAI-based framework for software vulnerability contributing factors assessment*, master's thesis, Concordia University, 2023.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] H. Q. Nguyen, T. Hoang, H. K. Dam, and A. Ghose, *Graph-based explainable vulnerability prediction*, Information and Software Technology, 176 (2024), p. 107386.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] M. Tufano, Y. Li, Z. Tu, and B. Ray, *Adaptive, reinforcement-learning-guided symbolic execution for smart contracts*, in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–13.

[35] United States President, *Executive order 14110: Safe, secure, and trustworthy development and use of artificial intelligence*. Federal Register, 88 FR 75191, Nov. 2023. Executive Order signed October 30, 2023, published November 1, 2023.

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

[37] 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.

[38] Y. Xu, Y. Zhang, C. Wang, S. Li, A. Zhang, and S. Deng, *ML-guided fuzzing: A systematic review*, in 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), 2023, pp. 577–588.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] B. Zhou, J. Li, X. Li, B. Hua, and J. Bao, *Leveraging on causal knowledge for enhancing the root cause analysis of equipment spot inspection failures*, Advanced Engineering Informatics, 54 (2022), p. 101799.

Appendix A

Algorithms and Pseudocode

This appendix records the core procedures that were referenced in the methodology: relation–aware GAT (Algorithm 1), the ACC constrained beam search for chain decoding (Algorithm 2), chain Selection and structural validation (Algorithm 3), and counterfactual intervention test (Algorithm 4).

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

Algorithm 1: Relation-aware GAT Message Passing (One Layer)

Input : Node states $\{\mathbf{h}_v^{(\ell)}\}$; typed neighbor sets $\mathcal{N}_r(v)$ for each relation $r \in \mathcal{R}$.
Output: Updated node states $\{\mathbf{h}_v^{(\ell+1)}\}$.
foreach *node* $v \in V$ **do**
  $\mathbf{m}_{\text{self}} \leftarrow \mathbf{W}_{\text{self}}\mathbf{h}_v^{(\ell)}$;
  $\mathbf{m}_{\text{sum}} \leftarrow \mathbf{0}$;
  foreach *relation* $r \in \mathcal{R}$ **do**
    foreach *neighbor* $u \in \mathcal{N}_r(v)$ **do**
      $\mathbf{z}_{uv}^{(r)} \leftarrow [\,\mathbf{W}_r\mathbf{h}_u^{(\ell)} \,\|\, \mathbf{W}_0\mathbf{h}_v^{(\ell)}\,]$;
      $e_{uv}^{(r)} \leftarrow \text{LeakyReLU}\big(\mathbf{a}_r^{\top}\mathbf{z}_{uv}^{(r)}\big)$;
    foreach *neighbor* $u \in \mathcal{N}_r(v)$ **do**
      $\alpha_{uv}^{(r)} \leftarrow \dfrac{\exp(e_{uv}^{(r)})}{\sum_{u' \in \mathcal{N}_r(v)} \exp(e_{u'v}^{(r)})}$;
      $\mathbf{m}_{\text{sum}} \leftarrow \mathbf{m}_{\text{sum}} + \alpha_{uv}^{(r)}\,\mathbf{W}_r\mathbf{h}_u^{(\ell)}$;
  $\mathbf{h}_v^{(\ell+1)} \leftarrow \text{ELU}\big(\mathbf{m}_{\text{self}} + \mathbf{m}_{\text{sum}}\big)$;
return $\{\mathbf{h}_v^{(\ell+1)}\}$;

Algorithm 2: ACC-constrained Beam Search for Chain Decoding

Input : Graph G; role probabilities $p_v^{\text{src/prop/san/sink}}$;
node scores s_v, chain scores z_v, edge compatibilities $c_{u\to v}^{(r)}$;
beam parameters (K, B, H);
predicates `cfg_ok`, `dfg_ok`, `ipa_ok`, `alias_ok`.

Output: Best admissible path $\pi^\star$ (root→propagation→sink) or `None`.

Path state. Each partial path π maintains: tip node v_t, taint footprint $T(\pi)$, call stack $C(\pi)$, accumulated guards $G(\pi)$, and score $S_{\text{ACC}}(\pi)$.

Initialization:

1. Select seed nodes $S_K \leftarrow \text{TopK}_v(s_v)$.
2. with $T(\pi), C(\pi), G(\pi)$ derived from v_0 initialize the beam as

$$\texttt{beams} \leftarrow \left\{ \pi = [v_0] \;\middle|\; v_0 \in S_K,\ S_{\text{ACC}}(\pi) = \log \sigma(s_{v_0}) \right\},$$

Expansion: for $t = 1$ **to** H **do**

- `pool` $\leftarrow \emptyset$;
- **foreach** *partial path* $\pi \in$ `beams` **do**
  - $u \leftarrow \text{tip}(\pi)$;
  - **foreach** *typed edge* $(u \xrightarrow{r} v)$ *in* G **do**
    - **if** `cfg_ok`$(u \to v)$ *is false* **or** `dfg_ok`$(u \to v, T(\pi))$ *is false* **or** `ipa_ok`$(u \to v, C(\pi))$ *is false* **or** `alias_ok`$(u \to v, T(\pi))$ *is false* **then**
      - **continue** to next neighbor;
    - `// Create successor path and update state`
    - $\pi' \leftarrow \pi \,\|\, v$; `// append` v
    - update $T(\pi'), C(\pi'), G(\pi')$ from $u \to v$;
    - $\Delta \leftarrow \alpha \log \sigma(z_v) + (1-\alpha)\, c_{u\to v}^{(r)}$;
    - $S_{\text{ACC}}(\pi') \leftarrow S_{\text{ACC}}(\pi) + \Delta$;
      $-\text{role\_penalty}(\pi') + \text{san\_bonus}(\pi')$;
      $-\text{len\_penalty}(\pi') - \text{rep\_penalty}(\pi')$;
      add π' to `pool`;
  - `// Keep top-`B `candidates by ACC score`
  - `beams` $\leftarrow \text{TopB}_{\pi \in \texttt{pool}} S_{\text{ACC}}(\pi)$;
  - **if** *some* $\pi \in$ `beams` *ends at a sink and no succ. can score higher* **then break**;
  - ;

Selection: Choose $\pi^\star$ as the highest-scoring valid path in `beams`, breaking ties in favour of: (i) fewer summary edges, (ii) inclusion of a sanitizer, and (iii) fewer distinct files.;

return $\pi^\star$;

Algorithm 3: Chain Selection and Structural Validation

Input : Candidate paths $\mathcal{P}$ from ACC decoding; thresholds $\tau_{\text{src}}, \tau_{\text{mid}}, \tau_{\text{sink}}$.
Output: Validated chain $\hat{\pi}$ or `None`.

Step 1: Role-shaped filtering.
foreach *path* $\pi = (v_0, \ldots, v_T) \in \mathcal{P}$ **do**
  if $p_{src}(v_0) < \tau_{src}$ **then**
    discard π and **continue**;
  if $\max_{t \in \{1,\ldots,T-1\}} \left(p_{prop}(v_t), p_{san}(v_t)\right) < \tau_{mid}$ **then**
    discard π and **continue**;
  if $p_{sink}(v_T) < \tau_{sink}$ **then**
    discard π and **continue**;
  keep π in the filtered set $\mathcal{P}'$;

Step 2: Interprocedural sufficiency.
if *the slice contains any CALL/ARG2PARAM/RET2* edges* **then**
  require that at least one path in $\mathcal{P}'$ uses an interprocedural edge; discard purely intra-procedural paths if interprocedural ones exist;

Step 3: Maximization.
Let $\hat{\pi}$ be the path in $\mathcal{P}'$ with the highest $S_{\text{ACC}}(\pi)$, breaking ties as in Appendix A.2 (fewer summary edges, includes a sanitizer, fewer files);

Step 4: Structural validation on $\hat{\pi}$.
Check that:

1. CFG is realizable end-to-end (including exceptional edges).
2. Any non-CFG hop is justified by DFG taint transport or accumulated guards.
3. The interprocedural stack is well-nested (push on CALL, pop on matching RET2*).
4. Alias consistency holds (non-empty points-to intersection for memory hops).

Step 5: Return.
if *all checks pass* **then**
  return $\hat{\pi}$;
else
  discard $\hat{\pi}$ and repeat Step 3 with the next-best path in $\mathcal{P}'$; if none remain, **return** `None`.

Algorithm 4: Counterfactual Intervention Test

Input : Graph slice of a positive example; recorded ACC hooks and baseline prediction p and chain score S_{ACC}.

Output : Probability change Δp and chain-score change ΔS_{ACC}.

Step 1: Choose an intervention. Select exactly one of:

1. Guard strengthening on a sanitizer that dominates the sink.
2. Sink neutralization (replace the dangerous sink with a benign equivalent).
3. Call unbinding (remove the ARG→PARAM edge that carries taint).

Step 2: Apply and re-run.

1. Modify the graph slice according to the chosen intervention.
2. Re-encode the slice and re-run ACC-constrained decoding.
3. Obtain the new prediction p' and chain score S'_{ACC}.

Step 3: Measure causal effect.

$$\Delta p \leftarrow p' - p, \qquad \Delta S_{\text{ACC}} \leftarrow S'_{\text{ACC}} - S_{\text{ACC}}.$$

return $(\Delta p, \Delta S_{\text{ACC}})$;

Appendix B

Model Equations and Training

This chapter centralizes the equations referenced by Section 3.2 and provides symbol glossaries.

B.1 GraphCodeBERT Feature Initialization: Equations

Token averaging. Given token embeddings $\mathbf{E} \in \mathbb{R}^{T\times 768}$ and the token index set $\mathcal{T}(i)$ aligned to node v_i,

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

Edge-aware token attention. Let $\phi(t) \in \mathbb{R}^k$ be local data–flow attributes for token t. With parameters $\mathbf{W}_e$ and $\mathbf{u}$:

$$\alpha_{i,t} \;=\; \frac{\exp\big(\mathbf{u}^\top \tanh\big(\mathbf{W}_e[\,\mathbf{E}_t \,\|\, \phi(t)\,]\big)\big)}{\sum_{s\in\mathcal{T}(i)} \exp\big(\mathbf{u}^\top \tanh\big(\mathbf{W}_e[\,\mathbf{E}_s \,\|\, \phi(s)\,]\big)\big)}, \tag{B.2}$$

$$\mathbf{x}_i^{\text{text}} \;=\; \sum_{t\in\mathcal{T}(i)} \alpha_{i,t}\, \mathbf{E}_t \in \mathbb{R}^{768}. \tag{B.3}$$

Projection and gated fusion. Let $\mathbf{x}_i^{\text{struct}} \in \mathbb{R}^{d_s}$ denote structural features:

$$\mathbf{h}_i^{\text{text}} \;=\; \mathrm{LN}\big(\mathbf{W}_t\, \mathbf{x}_i^{\text{text}}\big)\,, \qquad \mathbf{h}_i^{\text{struct}} \;=\; \mathrm{LN}\big(\mathbf{W}_s\, \mathbf{x}_i^{\text{struct}}\big)\,, \tag{B.4}$$

$$g_i \;=\; \sigma\Big(\mathbf{w}_g^\top \big[\,\mathbf{h}_i^{\text{text}} \,\|\, \mathbf{h}_i^{\text{struct}}\,\big] + b_g\Big)\,, \tag{B.5}$$

$$\mathbf{h}_i^{(0)} \;=\; g_i\, \mathbf{h}_i^{\text{text}} \;+\; (1-g_i)\, \mathbf{h}_i^{\text{struct}} \;\in\; \mathbb{R}^{d_0}. \tag{B.6}$$

Notation & Symbols for GraphCodeBERT

Symbol	Meaning
$\{t_1, \ldots, t_T\}$	Subword tokens for a node fragment; T=#tokens.
$\mathbf{E} \in \mathbb{R}^{T\times 768}$	GraphCodeBERT contextual token embeddings.
$\mathcal{T}(i)$	Token indices aligned to node v_i (by character span).
$\tilde{\mathbf{x}}_i^{\text{text}}$	Token average (Eq. B.1).
$\phi(t) \in \mathbb{R}^k$	Local attributes (def/use role, fan-in/out, etc.).
$\alpha_{i,t}$	Attention weight (Eq. B.2).
$\mathbf{x}_i^{\text{text}}$	Attention-pooled textual vector (Eq. B.3).
$\mathbf{x}_i^{\text{struct}}$	Structural features (type, degrees, SSA hints, literal stats).
$\mathbf{W}_t, \mathbf{W}_s$	Projections to width d_0 (Eq. B.4).
g_i	Scalar gate balancing text vs. structure (Eq. B.5).
$\mathbf{h}_i^{(0)}$	Fused node initialization (Eq. B.6).
d_0	Hidden width of first GAT layer.
L, S	Token window length and stride (typ. L=512, S=384).

B.2 Relation-Aware GAT: Equations

Typed attention and update. For $\mathcal{G} = (\mathcal{V}, \{\mathcal{E}_r\}_{r\in\mathcal{R}})$ and neighbors $\mathcal{N}_r(v)$:

$$\alpha_{u\to v}^{(r,\ell)} = \text{softmax}_{u\in\mathcal{N}_r(v)}\Big(\text{LeakyReLU}\big(a_r^\top [\, W_r^{(\ell)} h_u^{(\ell)} \,\|\, W_0^{(\ell)} h_v^{(\ell)} \,]\big)\Big), \tag{B.7}$$

$$h_v^{(\ell+1)} = \text{ELU}\left(W_{\text{self}}^{(\ell)} h_v^{(\ell)} + \sum_{r\in\mathcal{R}} \sum_{u\in\mathcal{N}_r(v)} \alpha_{u\to v}^{(r,\ell)}\, W_r^{(\ell)} h_u^{(\ell)}\right). \tag{B.8}$$

Node/seed/edge heads.

$$z_v = w_{\text{node}}^\top h_v^{(L)} + b_{\text{node}}, \qquad s_v = w_{\text{seed}}^\top h_v^{(L)} + b_{\text{seed}}, \tag{B.9}$$

$$c_{u\to v}^{(r)} = {h_u^{(L)}}^\top B_r\, h_v^{(L)} + \beta_r. \tag{B.10}$$

Seeds and path score.

$$\mathcal{S}_K = \text{TopK}\big(\{s_v : v \in \mathcal{V}\},\, K\big), \tag{B.11}$$

$$S(\pi) = \log\sigma(s_{v_0}) \;+\; \sum_{t=1}^{T} \big(\alpha\, \log\sigma(z_{v_t}) + (1-\alpha)\, c_{v_{t-1}\to v_t}^{(r_t)}\big). \tag{B.12}$$

Notation & Symbols for Relation-Aware GAT

Symbol	Meaning
$\mathcal{G} = (\mathcal{V}, \{\mathcal{E}_r\})$	Heterogeneous program graph; edges typed by $r \in \mathcal{R}$.
$\mathcal{R}$	Relation set (AST, CFG, DFG, CALL, ARG2PARAM, RET2CALL, RET2LHS, DFG_THIN).
$h_v^{(\ell)}$	Node state at layer ℓ; $h_v^{(0)}$ from fusion.
$\mathcal{N}_r(v)$	r-neighbors of node v.
$W_{\text{self}}^{(\ell)}, W_0^{(\ell)}, W_r^{(\ell)}$	Projections per layer/relation.
a_r	Relation-specific attention vector.
$\alpha_{u\to v}^{(r,\ell)}$	Attention weight (Eq. B.7).
z_v, s_v	Node vulnerability logit; seed score (Eq. B.9).
$c_{u\to v}^{(r)}$	Edge compatibility (Eq. B.10).
B_r, β_r	Bilinear form and scalar prior per relation.
$\mathcal{S}_K$	Top-K seeds by s_v (Eq. B.11).
$S(\pi)$	Log-additive path score (Eq. B.12).
α	Node/edge evidence mixing weight in $S(\pi)$.

B.3 Causal Knowledge Graph (CKG) Prior: Equation

Score mixture.

$$S^\star(\pi) \;=\; S(\pi) \;+\; \lambda\, S_{\text{CKG}}(\pi), \tag{B.13}$$

where $S_{\text{CKG}}(\pi)$ is a small prior bonus computed from unigram/bigram/trigram statistics mined from training chains; $\lambda \geq 0$ is a decoding-time weight.

Notation & Symbols for the CKG Prior

Symbol	Meaning
$\widehat{P}(r)$	Empirical unigram prior over relations from training chains.
$\widehat{P}(r_t \mid r_{t-1})$	Empirical bigram transition prior.
$S_{\text{CKG}}(\pi)$	Prior score computed from the relation sequence of π.
λ	Mixture weight in Eq. B.13.
$S^\star(\pi)$	Decoding score with CKG prior.

B.4 Adaptive Causal Contextualization (ACC): Equations

Feasibility predicates.

$$\mathsf{cfg\_ok}(u \to v) := \mathbf{1}\big[R_{\text{CFG}}(u, v) = 1\big], \tag{B.14}$$

$$\begin{aligned}\mathsf{dfg\_ok}(u \xrightarrow{r} v, \mathcal{T}(\pi)) \;:=\; & \mathbf{1}\big[(r = \texttt{DFG}) \wedge \mathsf{tainted}(u, \mathcal{T}(\pi))\big] \;\vee \\ & \mathbf{1}\big[\exists g \in \mathcal{G}(\pi) : v \text{ is control dependent on } g\big],\end{aligned} \tag{B.15}$$

$$\mathsf{ipa\_ok}(u \xrightarrow{r} v, C(\pi)) := \begin{cases} \text{push}(\text{callee}(v), \text{site}{=}u) & r = \texttt{CALL}, \\ \text{top of stack matches site of } u & r \in \{\texttt{RET2CALL}, \texttt{RET2LHS}\}, \\ 1 & \text{otherwise}, \end{cases} \tag{B.16}$$

$$\mathsf{alias\_ok}(u \xrightarrow{r} v, \mathcal{T}(\pi)) := \begin{cases} \mathbf{1}\big[\mathsf{pts}(u) \cap \mathsf{pts}(v) \neq \varnothing\big] & \text{if } r \text{ dereferences or writes}, \\ 1 & \text{otherwise}. \end{cases} \tag{B.17}$$

$$\mathsf{Adm}(u \xrightarrow{r} v \mid \pi) := \mathsf{cfg\_ok} \cdot \mathsf{dfg\_ok} \cdot \mathsf{ipa\_ok} \cdot \mathsf{alias\_ok}. \tag{B.18}$$

Role penalties and sanitizer bonus.

$$\mathsf{pen}_{\text{start}}(\pi) := \lambda_{\text{start}}\,[1 - p_{v_0}^{(\text{src})}]_+, \tag{B.19}$$

$$\mathsf{pen}_{\text{mid}}(\pi) := \lambda_{\text{mid}} \sum_{t=1}^{T-1} \Big(1 - \max\{p_{v_t}^{(\text{prop})}, p_{v_t}^{(\text{san})}\}\Big), \tag{B.20}$$

$$\mathsf{pen}_{\text{end}}(\pi) := \lambda_{\text{end}}\,[1 - p_{v_T}^{(\text{sink})}]_+, \tag{B.21}$$

$$\mathsf{pen}_{\text{role}}(\pi) := \mathsf{pen}_{\text{start}} + \mathsf{pen}_{\text{mid}} + \mathsf{pen}_{\text{end}}. \tag{B.22}$$

$$\mathsf{bonus}_{\text{san}}(\pi) := \mu \sum_{s \in \pi} \mathbf{1}\big[\text{Dom}(s, \text{sink})\big] \cdot \mathbf{1}\big[\mathsf{affects\_taint}(s, \pi)\big]. \tag{B.23}$$

ACC objective and CKG mixture.

$$\mathsf{pen}_{\text{len}}(\pi) := \eta\,\text{len}(\pi), \tag{B.24}$$

$$\mathsf{pen}_{\text{rep}}(\pi) := \rho\,\text{rep}(\pi), \tag{B.25}$$

$$S_{\text{ACC}}(\pi) := S(\pi) - \mathsf{pen}_{\text{role}}(\pi) + \mathsf{bonus}_{\text{san}}(\pi) - \mathsf{pen}_{\text{len}}(\pi) - \mathsf{pen}_{\text{rep}}(\pi), \tag{B.26}$$

$$S^{\star}_{\text{ACC}}(\pi) \;=\; S_{\text{ACC}}(\pi) \;+\; \lambda\, S_{\text{CKG}}(\pi). \tag{B.27}$$

Notation & Symbols for ACC

Symbol	Meaning
$\pi = (v_0, \ldots, v_T)$	Candidate path (chain).
$\mathcal{T}(\pi), C(\pi), \mathcal{G}(\pi)$	Taint footprint, call stack, accumulated guards.
Predicates	cfg_ok, dfg_ok, ipa_ok, alias_ok (Eqs. B.14–B.17).
$\mathsf{Adm}(\cdot)$	Conjunctive admissibility (Eq. B.18).
$p_v^{(\text{src/prop/san/sink})}$	Role probabilities.
$\lambda_{\text{start}}, \lambda_{\text{mid}}, \lambda_{\text{end}}$	Role penalty weights.
μ	Sanitizer bonus weight.
η, ρ	Length and repetition penalty weights.
$S_{\text{ACC}}(\pi), S^{\star}_{\text{ACC}}(\pi)$	ACC scores without/with CKG (Eqs. B.26, B.27).

B.5 CKG Prior Scoring

Let $\pi(i \xrightarrow{r} j)$ be the mined prior over relation motifs from the training split. With smoothing ϵ and temperature τ,

$$\tilde{\pi}(i \xrightarrow{r} j) = \frac{\pi(i \xrightarrow{r} j) + \epsilon}{\sum_{k \in N(i)} \left(\pi(i \xrightarrow{r_k} k) + \epsilon\right)}, \qquad \pi_\tau(i \xrightarrow{r} j) = \frac{\tilde{\pi}(i \xrightarrow{r} j)^{1/\tau}}{\sum_{k \in N(i)} \tilde{\pi}(i \xrightarrow{r_k} k)^{1/\tau}}.$$

During decoding, admissible edges are scored by mixing ACC/beam score $s(i \xrightarrow{r} j)$ with the prior:

$$S_{t+1} \;=\; S_t \;+\; s(i \xrightarrow{r} j) \;+\; \lambda \, \log \pi_\tau(i \xrightarrow{r} j),$$

where λ is the prior weight, ϵ the add-one smoothing, and τ the temperature. Numeric settings are listed in Appendix. D.1.

B.6 Chain Extraction and Validation: Equations

Role-shaped validity criteria.

$$p_{v_0}^{(\text{src})} \;\geq\; \tau_{\text{src}}, \tag{B.28}$$

$$\exists\, t \in \{1, \ldots, T-1\} : \; \max\left(p_{v_t}^{(\text{prop})}, \, p_{v_t}^{(\text{san})}\right) \;\geq\; \tau_{\text{mid}}, \tag{B.29}$$

$$p_{v_T}^{(\text{sink})} \;\geq\; \tau_{\text{sink}}. \tag{B.30}$$

Selection.

$$\hat{\pi} \;=\; \arg\max_{\pi \in \mathcal{P}_{\text{adm}}} \; S_{\text{ACC}}(\pi). \tag{B.31}$$

Notation & Symbols for Chain Extraction

Symbol	Meaning
$\tau_{\text{src}}, \tau_{\text{mid}}, \tau_{\text{sink}}$	Thresholds for start/middle/end criteria (Eqs. B.28–B.30).
$\mathcal{P}_{\text{adm}}$	Set of admissible paths under ACC constraints.
$\hat{\pi}$	Selected chain (Eq. B.31).

B.7 Training Objective: Equation

Composite loss.

$$\mathcal{L} = \text{BCE}\left(\sigma(\mathbf{z}), \, \hat{y}\right) + \lambda_{\text{flow}}\mathcal{L}_{\text{flow}} + \lambda_{\text{cf}}\mathcal{L}_{\text{cf}} + \lambda_{\text{ent}}\mathcal{L}_{\text{ent}} + \lambda_{\text{spec}}\mathcal{L}_{\text{spec}} + \lambda_{\text{san}}\mathcal{L}_{\text{san}}. \tag{B.32}$$

Notation & Symbols for Training Objective

Symbol	Meaning
$\hat{y} \in \{0, 1\}$	Graph label (vulnerable / non-vulnerable).
$\mathbf{z} \in \mathbb{R}$	Graph logit (pooled over final node states).
BCE	Binary cross-entropy.
$\mathcal{L}_{\text{flow}}$	Flow consistency (upscores chain edges/paths, downscores distractors).
$\mathcal{L}_{\text{cf}}$	Counterfactual consistency (confidence drop on minimal edits).
$\mathcal{L}_{\text{ent}}$	Attention entropy (encourages decisive attention).
$\mathcal{L}_{\text{spec}}$	Spectral norm control (stability).
$\mathcal{L}_{\text{san}}$	Sanitizer alignment (when sanitizer dominates sink).
$\lambda_\star \geq 0$	Weights for the corresponding regularizers.

Appendix C

Metrics and Diagnostics

C.1 Calibration and Thresholding

Operating Points Validation-optimal F1 threshold $\tau_{\mathrm{F1}^\star}$ and fixed threshold τ=0.5 are used for reporting.

Expected Calibration Error (ECE)

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

S_b are prediction bins, N is the number of examples.

C.2 Standard Classification Metrics

$$\mathrm{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}}, \tag{C.2}$$

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \tag{C.3}$$

$$\mathrm{F1} = \frac{2\,\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}, \tag{C.4}$$

$$\mathrm{MacroF1} = \tfrac{1}{2}\left(\mathrm{F1}_{\mathrm{pos}} + \mathrm{F1}_{\mathrm{neg}}\right), \quad \mathrm{MicroF1} = \frac{2\sum \mathrm{TP}}{2\sum \mathrm{TP} + \sum \mathrm{FP} + \sum \mathrm{FN}}. \tag{C.5}$$

AUROC and AUPRC are computed on raw scores.

C.3 Chain-Centric Metrics

Validity of Predicted Chains

$$\text{ValidityRate} = \frac{\#\{\text{predicted chains passing all ACC checks}\}}{\#\{\text{predicted chains}\}}. \tag{C.6}$$

Structural Fidelity Let predicted chain nodes/edges be $(\hat{V}, \hat{E})$ and ground truth $(V^\star, E^\star)$.

$$\text{NodeCov} = \frac{|\hat{V} \cap V^\star|}{|V^\star|}, \qquad \text{EdgeCov} = \frac{|\hat{E} \cap E^\star|}{|E^\star|}, \tag{C.7}$$

$$\text{LCS Ratio} = \frac{\text{LCS}(\hat{\pi}, \pi^\star)}{|\pi^\star|}, \tag{C.8}$$

$$\text{CO}(\alpha) = \alpha \cdot \text{NodeCov} + (1-\alpha) \cdot \text{EdgeCov}, \quad \alpha \in [0,1], \tag{C.9}$$

$$\text{RoleCov}_r = \frac{|\hat{V}_r \cap V_r^\star|}{|V_r^\star|}, \quad r \in \{\text{src,san,prop,sink}\}. \tag{C.10}$$

Interprocedurality (IPA)

$$\text{IPA Rate} = \frac{\#\{\hat{\pi} \text{ using CALL/ARG} \to \text{PARAM/RET} \to \text{CALLER or LHS}\}}{\#\{\text{predicted chains in slices containing such edges}\}}. \tag{C.11}$$

C.4 Counterfactual Metrics

Counterfactual Consistency Score (CCS) For applicable graphs $i \in \mathcal{I}$, with original p_i and intervened p_i^{do},

$$\text{CCS}_i = (p_i - p_i^{\text{do}})^2, \tag{C.12}$$

$$\text{CCS} = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \text{CCS}_i. \tag{C.13}$$

Directional consistency uses $s_i \in \{+1, -1\}$:

$$\Delta_i^{\text{dir}} = s_i\,(p_i - p_i^{\text{do}}), \qquad \text{DCR} = \frac{1}{|\mathcal{I}|} \sum_i \mathbb{1}\{\Delta_i^{\text{dir}} > 0\}. \tag{C.14}$$

Causal Feature Attribution Measure (CFAM) Let on-chain features F_c and off-chain F_s with attributions $A_i(f) \geq 0$,

$$\text{CFAM}_i = \frac{\sum_{f \in F_c} |A_i(f)|}{\sum_{f \in F_c \cup F_s} |A_i(f)|}, \qquad \text{CFAM} = \frac{1}{N} \sum_i \text{CFAM}_i. \tag{C.15}$$

Attributions are normalized per graph.

C.5 Decoding diagnostics

For completeness, I report the auxiliary decoding diagnostics used in the artifact. The Chain Success Rate (CSR), Admissible Expansion Ratio (AER), and Prior Influence Rate (PIR) are defined as:

$$\mathrm{CSR} = \frac{\#\{\text{positives with a nonempty chain}\}}{\#\{\text{positives}\}}, \tag{C.16}$$

$$\mathrm{AER} = \frac{\#\{\text{candidate edges admitted by ACC}\}}{\#\{\text{candidate edges considered}\}}, \tag{C.17}$$

$$\mathrm{PIR} = \frac{\#\{\text{steps where the CKG prior changes the winning rank}\}}{\#\{\text{decoding steps}\}}. \tag{C.18}$$

Average chain length, interprocedural hop ratio, motif coverage, and the ACC rejection mix (CFG/DFG/IPA/alias) are summarized as descriptive statistics in the released logs, but are not part of the main quantitative analysis in this research.

Appendix D

Extended Results and Settings

This appendix centralizes configuration details, intermediate diagnostics, and practical instructions to reproduce the reported tables and figures.

D.1 Hyperparameters and Environment

Recommended defaults

K=8, B=24, H=5, α=0.7, λ=0.2, ϵ=10^{-3}, τ=1.0, motif top-K=500.

Model and optimization settings

Table D.1 summarizes the model architecture, optimization hyperparameters, and training configuration used in all experiments.

Software and hardware manifest

Python version, key libraries, and a lockfile with package hashes are included with the artifact for exact replay.

D.2 Beam Search and ACC Diagnostics

The table D.2 summarizes per-graph decoding behavior under the default settings (K=8, B=24, H=5, α=0.7). Values are medians with P90 where noted.

Beam Sensitivity (K, B, H)

Table D.3 reports validation metrics and per-graph latency for the GCBERT+Struct model across beam sizes.

Table D.1: Model architecture and optimization settings.

Component	Setting
GAT depth / width	L=3 layers; hidden width d=64; one 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; lr 2×10^{-3}; wd 10^{-4}; cosine decay (40 epochs), 2-epoch warmup
Gradient control	Global norm clip 1.0; mixed precision (FP16 projections/attention, FP32 accumulation)
Aux losses (if enabled)	λ_{flow}=0.5, margin 0.5; λ_{cf}=0.5, margin 0.7; λ_{ent}=0.05; λ_{spec}=10^{-4}
ACC / beams	Seeds K=8; 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 prob. 0.3)
Language model	GraphCodeBERT frozen; 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

Table D.2: Beam and ACC diagnostics under K=8, B=24, H=5 (per graph).

Quantity	Median	P90	Struct	GCBERT+Struct
Seeds used (of 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 bonus	41.6%	51.0%	39.8%	43.2%

Table D.3: Beam sensitivity (Valid, GCBERT+Struct; latency in ms per graph).

K	B	H	Validity	IPA	LCS	Latency
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

Table D.4: Decoder diagnostics for rare cases and suggested remedies.

Symptom	Likely cause	Signal	Remedy
Early termination	True path exceeds H or detours	Low score near t=H, no successors	Increase H or add thin DFG summaries
No interprocedural hop	Source or sink API missing from lexicon	High node score, zero `ARG2PARAM/RET2*`	Extend lexicon or enable API summaries
Alias mismatch	Coarse points-to for container or pointer hop	`alias_ok` rejects edge	Refine buckets; add container-aware rules
Guard over-credit	Guard does not dominate the sink region	Bonus lifts wrong branch	Tighten dominance; require taint effect
Rare relation down-weighted	Prior penalizes uncommon pattern	Prior penalty visible in path deltas	Reduce mixture λ or smooth with larger K

Decoder Diagnostics and Remedies: Table D.4 lists rare decoding issues, their signals, and practical fixes.

D.3 Additional Diagnostics

Reported items

- Per-hop rank traces with and without the CKG prior.
- ACC rejection histograms: CFG, def–use/guard, IPA stack, alias conflict.
- Motif coverage@K and interprocedural hop ratio distributions.
- Admissible expansion ratio curves versus decoding depth.

Appendix E

Dataset Card and Licensing

Name and scope: Experiments use *ReposVul*, a repository-level corpus that links CVE/CWE metadata suitable for interprocedural, chain-centric analysis.

Composition: This dissertation uses the C/C++ subset with the default splits. Each entry contains a code snapshot, with aligned views at line, function, file, and repository level, and 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 filtering of outdated patches.

Labels and quality: Vulnerability labels follow the original ReposVul patch pairs. Mechanism roles (source, sanitizer, propagator, sink) are assigned by rules and consistency checks.

Splits and leakage safeguards: Default train/validation/test splits are used without modification. Projects do not cross partitions, parent/child patches are never split, and identical files/CVEs are removed across splits to reduce leakage.

Intended use: The dataset is used for repository-level vulnerability modeling, with an emphasis on interprocedural chain reconstruction and interpretability.

Licensing and attribution: Use of *ReposVul* follows the terms specified by its authors.

Known limitations: Not all vulnerabilities admit a single clear executable chain, and some fixes are entangled with refactoring.

Appendix F

Role Lexicon and Pattern Rules

This appendix lists the APIs, idioms, and structural patterns used to assign *source*, *sanitizer*, *propagator*, and *sink* roles (Section 3.1.4).

API Families and Examples

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

Structural Patterns

1. Guards that dominate the sink and constrain tainted values (e.g., index/length checks on paths from source to sink).
2. Def–use chains that cross calls via actual/formal bindings (ARG→PARAM; RET→CALLER/RET→LHS).
3. Alias flows through pointers, references, or containers where points-to sets intersect along writes and reads.

Precedence: If multiple roles apply to a node, the precedence is *sanitizer* > *sink* > *propagator* > *source*, resolved using CFG dominance and DFG reachability (see Chapter 3).

Appendix G

Reproducibility and Artifact

This appendix summarizes the released artifact and how it supports end-to-end reproducibility, from data preparation to model training and evaluation.

G.1 Data provenance

Each split ships a manifest with repository names, commit hashes, file paths, and per-example checksums. Parent/child commit IDs are recorded so vulnerable and fixed snapshots can be reconstructed from the original repositories.

G.2 Graph construction

The graph builder configuration includes parser versions, normalization options, and enabled relation families: AST/CFG/DFG, `CALL`, `ARG2PARAM`, `RET2CALL`, `RET2LHS`, and alias summaries.

G.3 Embeddings and caches

The artifact records the GraphCodeBERT checkpoint and tokenizer IDs, maximum sequence length and stride, and normalization flags.

G.4 Training and decoding configuration

Versioned configuration files specify encoder and decoder settings, including learning rate, schedule, weight decay, dropout, gradient clipping, loss weights, beam parameters (K, B, H, α), augmentation probabilities, and early stopping criteria. ACC and CKG parameters are included so both training and decoding behaviour can be reproduced.

G.5 Environment and determinism

Experiments use PyTorch 2.4.1 with CUDA 12.1 on a single GPU. Random seeds are fixed, deterministic and cuDNN flags are set where possible, and mixed precision uses dynamic loss scaling with critical reductions in FP32. An environment manifest (lockfile and package hashes) is included.

G.6 Artifact contents

The released artifact contains:

- Per-graph predictions and reconstructed chains (with beam traces).
- Calibration parameters and ECE bin assignments.
- Counterfactual intervention logs (pre/post probabilities and edit types).
- Configuration files for all reported experiments.
- Environment manifests and package hashes.
- CKG prior data and motif statistics.
- Decoding diagnostics (per-hop scores and ACC rejection reasons).
- Scripts to rebuild graphs, train models, export chains, and recompute all reported metrics and figures.

G.7 Project layout

```
Thesis-causal-vul/
|-- WORKPLAN.md
|-- README.md
|-- commands.txt
|-- ScripstList.txt
|-- requirements.txt
|-- notebooks/
|   |-- 01_ReposVul.ipynb
|   |-- 02_word2vec_training.ipynb
|   |-- 03_causal_chain_demo.ipynb
|   `-- colab.ipynb
|-- tools/
|   |-- export_pdg_env.sc
|   `-- validate_pdg_dir.py
```

```
|-- Report/
|   |-- Causalchain_pretty.png
|   |-- Causalchain_raw.png
|   |-- eval_colab/
|   |   |-- eval_report.json
|   |   `-- test_preds.csv
|   `-- ReposVul_report/
|       |-- ReposVul_CCPP_Full_Report.md
|       `-- _edges/
|           |-- test_c_cpp_repository2__caller_func_callee.csv
|           |-- train_c_cpp_repository2__caller_func_callee.csv
|           `-- valid_c_cpp_repository2__caller_func_callee.csv
`-- src/
|-- __init__.py
|-- build_dataset_jsonl.py
|-- infer_one_slice_gcbert.py
|-- infer_one_slice_pretty.py
|-- step4_train_gnn_attn.py
|-- step4_train_gnn_attn_v1.py
|-- step5_export_chains.py
|-- step6_eval_report.py
|-- train.py
|-- preprocess/
|   |-- __init__.py
|   |-- embed_lines_gcbert.py
|   |-- slices_to_pyg_gcbert.py
|   |-- step1_prepare_diversevul.py
|   |-- step2_generate_pdg.py
|   |-- step2c_pipeline_validate_and_slice.py
|   |-- transformer_cache.py
|   `-- external/
|       |-- program_slice.py
|       |-- sensiAPI.txt
|       |-- sensiAPI_A.txt
|       `-- sensiAPI_B.txt
`-- utils/
|-- hashing.py
`-- pdg_io.py
```

G.8 Step-by-step reproducibility guide

Prerequisites

- Python 3.10+ and a CUDA-enabled GPU (for CPU-only, training and decoding will be slower).
- Optional: Joern installed or available on PATH if rebuilding CPG/PDG from raw code.

1. Install dependencies

Linux

```
python -m pip install -r requirements.txt
```

Windows PowerShell

```
python -m pip install -r .\requirements.txt
```

2. Prepare datasets and program graphs

From raw sources (ReposVul file lists), run the preprocessing pipeline; **Linux**

```
python src/preprocess/step1_prepare_diversevul.py
python src/preprocess/step2_generate_pdg.py
python src/preprocess/step2c_pipeline_validate_and_slice.py
```

Windows PowerShell

```
python .\src\preprocess\step1_prepare_diversevul.py
python .\src\preprocess\step2_generate_pdg.py
python .\src\preprocess\step2c_pipeline_validate_and_slice.py
```

3. Build or load GraphCodeBERT caches

Linux

```
python src/preprocess/embed_lines_gcbert.py
```

Windows PowerShell

```
python .\src\preprocess\embed_lines_gcbert.py
```

Then convert slices to PyTorch Geometric hetero graphs: **Linux**

```
python src/preprocess/slices_to_pyg_gcbert.py
```

Windows PowerShell

```
python .\src\preprocess\slices_to_pyg_gcbert.py
```

4. Assemble training JSONL (if required)

Linux

```
python src/build_dataset_jsonl.py
```

Windows PowerShell

```
python .\src\build_dataset_jsonl.py
```

5. Train the model

Linux

```
python src/step4_train_gnn_attn.py
# or:
python src/step4_train_gnn_attn_v1.py
```

Windows PowerShell

```
python .\src\step4_train_gnn_attn.py
# or:
python .\src\step4_train_gnn_attn_v1.py
```

6. Export chains on the test split

Linux

```
python src/step5_export_chains.py
```

Windows PowerShell

```
python .\src\step5_export_chains.py
```

7. Compute metrics and reports

Linux

```
python src/step6_eval_report.py
```

Windows PowerShell

```
python .\src\step6_eval_report.py
```

This produces summary files under `Report/` such as:

```
Report/eval_colab/eval_report.json
Report/eval_colab/test_preds.csv
```

8. Optional: pretty print a single slice

Linux

```
python src/infer_one_slice_pretty.py
```

Windows PowerShell

```
python .\src\infer_one_slice_pretty.py
```

G.9 Determinism and validation checks

Seeds are fixed in configs; decoding is deterministic given identical environment and seeds. Checksums verify inputs and expected summaries. A quick sanity script confirms: (i) project-disjoint splits, (ii) at least one valid chain on known positives, and (iii) improved calibration (lower ECE) after validation temperature scaling.

G.10 Source for Figure 4.1 and Chain Trace

The files below reproduce Figure 4.1. Decoding used ACC with K=8, B=24, H=5, and α=0.7; the CKG prior weight was λ=0.2. Paths are relative to the project root. The exact decoded chain trace is included for reference.

app/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* stage_clean(char* in);
char* stage_route(char* in);
void stage_execute(char* cmd);

int main(int argc, char** argv) {
  char buf[256];
  size_t n;

  memset(buf, 0, sizeof(buf));
  if (fgets(buf, sizeof(buf), stdin) == NULL) {
    return 0;
  }

  n = strlen(buf);
  if (n > 0 && (buf[n - 1] == '\n' || buf[n - 1] == '\r')) {
    buf[n - 1] = '\0';
```

```
  }

  char* s1 = stage_clean(buf);
  char* s2 = stage_route(s1);
  stage_execute(s2);
  return 0;
}
```

lib/shell.c

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>

  static char route_buf1[512];
  static char route_buf2[512];
  static char route_buf3[512];
  static char exec_local[512];

  static char* build_prefix(char* in) {
    snprintf(route_buf1, sizeof(route_buf1), "runner␣%s", in);
    return route_buf1;
  }

  static char* add_route_flags(char* in) {
    snprintf(route_buf2, sizeof(route_buf2),
                         "%s␣--channel=cli␣--retry=0", in);
    return route_buf2;
  }

  static char* finalize_route(char* in) {
    snprintf(route_buf3, sizeof(route_buf3), "%s␣--complete", in);
    return route_buf3;
  }

  char* stage_route(char* in) {
    char* a = build_prefix(in);
    char* b = add_route_flags(a);
    char* c = finalize_route(b);
    return c;
  }

  void stage_execute(char* cmd) {
    size_t n;
    if (cmd == NULL) {
      return;
```

```
  }
  n = strlen(cmd);
  if (n >= sizeof(exec_local)) {
    n = sizeof(exec_local) - 1;
  }
  memset(exec_local, 0, sizeof(exec_local));
  strncpy(exec_local, cmd, n);
  exec_local[n] = '\0';
  system(exec_local); /* external command execution */
}
```

Decoded chain trace for Case A

```
Root -> ... -> Sink (hops=7)

[ROOT] if (fgets(buf, sizeof(buf), stdin) == NULL) {
  (CALL, app/main.c, line 15, p=0.475)
  `-- [CFG] exact=CFG
  [->] return 0;
  (BLOCK, app/main.c, line 16, p=0.470)
  `-- [CFG] exact=CFG
  [->] }
(BLOCK, app/main.c, line 17, p=0.528)
`-- [CFG] exact=CFG
[->] n = strlen(buf);
(BLOCK, app/main.c, line 19, p=0.472)
`-- [CFG/DFG] exact=CFG,DFG
[->] if (n > 0 && (buf[n - 1] == '\n' || buf[n - 1] == '\r')) {
  (CALL, app/main.c, line 20, p=0.395)
  `-- [CPG] exact=CALL
  [->] n = sizeof(exec_local) - 1;
  (BLOCK, lib/shell.c, line 40, p=0.524)
  `-- [DFG] exact=DFG
  [->] exec_local[n] = '\0';
  (BLOCK, lib/shell.c, line 44, p=0.451)
  `-- [CFG] exact=CFG
  [SINK] system(exec_local); /* external command execution */
  (CALL, lib/shell.c, line 45, p=0.445)
  `-- [CPG] exact=CALL
```

The same trained checkpoint and decoding settings were used for all case studies.