Context-Driven Root Cause Analysis for Software Security Vulnerabilities

by

Md Iqbal Hossain Shuvo

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

Bishop's University
Canada
November 2025

Abstract

Software vulnerabilities in large systems rarely stem from a single line of code. Instead, they emerge from chains of decisions and data flows that cross multiple functions, files, and modules. Existing tools, including both traditional static and dynamic analysis and modern learning-based detectors, are good at flagging suspicious locations but rarely explain how a vulnerability actually arises and propagates through a program. As a result, their predictions are often brittle, difficult to trust, and only weakly connected to the steps a developer must take to fix the underlying problem.

This thesis addresses this limitation by treating the entire path from the initial cause of a vulnerability to its exploitable endpoint as the main object of analysis. The central goal is to reconstruct *executable interprocedural root-to-sink vulnerability chains* that a developer can follow across functions and files. To achieve this, the work first builds a program representation that records how control and data move within and between functions in real software projects. On top of this representation, a learning-based model is trained not only to decide whether a piece of code is vulnerable, but also to identify and assemble the sequence of steps that carry the vulnerable state from its origin to a sensitive operation.

Beyond simply producing these chains, the thesis introduces criteria to check whether the explanations are truly aligned with the model's behaviour. These criteria probe how predictions change when key parts of a chain are edited and measure how much of the model's attention falls on the reconstructed path. An empirical study on real-world repositories shows that focusing on interprocedural chains leads to more stable predictions, clearer explanations, and guidance that is better suited to debugging and remediation than conventional "black-box" scores. In this way, the thesis moves vulnerability detection closer to tools that do not just say *where* a problem is, but explain *how* and *why* it can be exploited.

Acknowledgments

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software vulnerability detection remains a critical yet complex challenge in software engineering. 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, contemporary codebases span millions of lines of code and are developed by distributed teams over long periods of time. This growth in complexity increases architectural coupling and the likelihood that subtle defects persist into production. When such defects are exploitable, they become security vulnerabilities that threaten confidentiality, integrity, and availability, and their effects may propagate across dependent components and services [48, 64, 47].

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 [36, 11, 51]. *Dynamic analysis* executes programs with concrete or instrumented inputs to observe runtime behaviour directly, often using monitoring, tracing, or sandboxing to detect anomalous states [62, 8]. 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 [48, 47].

Recent years have seen a substantial shift toward *data-driven approaches* for software vulnerability detection. These methods, which include machine learning and deep learning based models, aim to automate vulnerability discovery by learning patterns from code representations such as token sequences, abstract syntax trees, control-flow graphs, and data-flow graphs. Graph-based models align naturally

with program structure: Abstract Syntax Trees (ASTs) capture syntactic hierarchies, Control-Flow Graphs (CFGs) encode possible execution paths, and Data-Flow Graphs (DFGs) represent value dependencies. These complementary views can be merged into unified program representations such as the Code Property Graph (CPG) to support integrated reasoning over syntax, control, and data [63]. Building on this foundation, models such as *Devign* and later graph neural network variants have achieved strong benchmark performance by fusing information from these structural perspectives [31]. Post hoc explainers such as *LEMNA* and *GNNExplainer* highlight influential features or subgraphs to clarify model decisions [18, 65]. Structure-aware pretraining models such as *GraphCodeBERT* further enrich token embeddings with data-flow semantics and have been shown to improve downstream vulnerability detection [17].

Despite this progress, many existing data-driven detectors remain *correlation-centric*. Traditional deep learning models tend to rely heavily on statistical associations in training data, often capturing superficial patterns rather than the underlying causes of vulnerabilities. They typically focus on isolated statements or small subgraphs that are predictive of the vulnerability label and emphasize *where* a flaw is likely to occur, rather than *how* it emerges and propagates through interacting control and data flows [48, 64]. This correlation-centric bias produces several limitations. Models can become sensitive to dataset-specific artefacts, such as naming conventions or formatting, and may fail under benign transformations such as refactoring, formatting changes, or identifier renaming [48, 64]. Their predictions also offer limited interpretability for developers, since they seldom reconstruct the broader execution context that links root causes, intermediate propagation, and exploitable sinks [4, 24, 44, 9].

These limitations are particularly severe for *interprocedural* vulnerabilities. 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. Modeling such behaviours requires reasoning not only about local syntax and intra-procedural structure but also about interprocedural semantics, including call and return edges, argument to parameter bindings, return to caller relationships, and conservative aliasing across function and module boundaries [47]. Existing static, dynamic, fuzzing, and data-driven techniques typically approximate these patterns using limited context or heuristics and rarely construct explicit, executable chains that show how vulnerable state propagates through the system [48, 47].

To address these challenges, this thesis adopts a *causal and context-aware* perspective on vulnerability detection. In this work, *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 [63, 17]. 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. Capturing this semantic context is essential for reconstructing meaningful explanations of vulnerability behaviour, especially in interprocedural scenarios [47]. At the same time, the thesis 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, 24, 44, 9]. In the context of vulnerability detection, embedding causal reasoning into program analysis is not only attractive, but necessary, since remediation decisions depend on understanding which code elements and propagation paths are actually responsible for a vulnerability.

These limitations motivate a shift from purely correlational pattern recognition to *causal, context-aware, interprocedural reasoning* in vulnerability detection [4, 24, 44, 9]. In this thesis, context awareness refers to the model's ability to understand not only the syntactic structure of source code but also its semantic meaning, including how different code components interact through control flow, data dependencies, and functional relationships. Vulnerabilities often arise from complex interactions that span multiple functions, modules, or execution paths rather than from isolated statements [47]. Consequently, there is a need for an approaches that construct structured causal chains, tracing the origin and spread of vulnerabilities across interprocedural boundaries. By embedding semantic knowledge of program behavior and focusing on causal relationships instead of surface-level correlations, such approaches can provide deeper, more trustworthy, and more actionable explanations of how vulnerabilities emerge and propagate in real software systems [48, 64, 47, 4, 24, 44, 9].

1.1 Problem Definition and Scope

Software vulnerabilities rarely arise from a single, isolated statement; instead, they often emerge from interactions that span multiple functions, files, and modules in large, evolving codebases. Tainted inputs 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. Capturing such behaviours requires reasoning about both intra-procedural structure and interprocedural semantics, including how data and control flow across function boundaries, files, and modules [47].

Existing vulnerability detection techniques, including static and dynamic analysis, fuzzing and penetration testing, and modern data-driven detectors, provide important but incomplete coverage of this interprocedural behaviour. Classical static, dynamic, and fuzzing-based solutions are effective at finding local defects or

triggering specific failure scenarios, but they rarely reconstruct *explicit, executable root-to-sink vulnerability chains* that connect initial sources, intermediate propagation steps, and final sinks across procedural boundaries. Similarly, most data-driven detectors are trained to classify localized code regions or limited graph neighborhoods and tend to focus on pointwise predictions of *where* a vulnerability may occur, rather than modelling *how* vulnerable state propagates across functions and modules in a causally meaningful way [48, 64, 47]. As a result, their explanations are often fragmentary, correlation-driven, and difficult to validate against actual execution behaviour [4, 24, 44, 9].

This thesis addresses this gap by treating the full *interprocedural root→propagation →sink chain* as the primary object of explanation. The goal of this thesis is to develop a context aware data-driven causal approach that reconstructs *interprocedural root-to-sink vulnerability propagation chains* in source code, aligning vulnerability predictions with executable, causally grounded explanations of how vulnerabilities arise and propagate across function and module boundaries.

1.2 Research Objectives and Questions

To achieve the goal of reconstructing executable interprocedural root-to-sink vulnerability propagation chains, this thesis pursues the following research objectives.

Research Objectives

Each objective is grounded in an interprocedural, chain-centric causal reasoning paradigm explicitly modeling how vulnerabilities emerge, propagate, and compound across procedure boundaries.

O1 - Interprocedural representation: Design a program representation that captures interprocedural control and data dependencies, combining structures such as Abstract Syntax Trees (ASTs), Control-Flow Graphs (CFGs), Data-Flow Graphs (DFGs), and Program/Code Property Graphs (PDG/CPG), in a form suitable for root-to-sink causal analysis across functions, files, and modules.

O2 - Causal chain modelling: Create a constraint-based approach to generate executable, interprocedural vulnerability chains, explicitly representing control flow, data dependencies, and aliasing to produce chains that are viable, minimal, and mechanically valid.

Research Questions

Building on the above objectives, this thesis investigates the following research questions:

RQ1 – Interprocedural representation: How can a program representation be designed that faithfully captures interprocedural control and data dependencies (including AST, CFG, DFG, and PDG/CPG views) in a form that supports executable, root-to-sink causal analysis across functions, files, and modules?

RQ2 – Interprocedural causal chain construction: How can a context-aware causal reasoning approach systematically construct executable interprocedural vulnerability chains that are viable, minimal, and consistent with the underlying control flow, data dependencies, and aliasing relationships?

1.3 Methodology Overview

This thesis constructs an interprocedural program graph from real-world vulnerable repositories and then applies chain-centric causal reasoning over this graph. The codebase is parsed into structured representations such as Abstract Syntax Trees, Control-Flow Graphs, and Data-Flow Graphs, which are integrated into a unified Program/Code Property Graph. This graph is augmented with interprocedural semantics, including call edges, argument→parameter mappings, return→caller links, and conservative aliasing, so that paths from candidate sources to security-sensitive sinks can be followed across functions, files, and modules. Nodes in this heterogeneous graph are embedded using a structure-aware pretrained encoder (for example, GraphCodeBERT) in combination with explicit structural features, yielding rich contextual representations that reflect both local syntax and non-local control and data dependencies.

On top of these embeddings, the thesis employs a graph-based model equipped with Attention and Adaptive Causal Contextualization (ACC) to identify and assemble executable interprocedural causal chains. Causal attention highlights code elements that exert causal influence on vulnerability outcomes, while ACC propagates and refines these signals along control-flow and data-flow edges, allowing the model to trace root-to-sink propagation across function boundaries and to construct concise, executable chains for each vulnerable slice. The resulting chains are then assessed using causal validation criteria, such as the Counterfactual Consistency Score (CCS) and the Causal Feature Attribution Measure (CFAM), which complement conventional classification metrics and are treated as part of the evaluation strategy rather than as standalone research objectives.

1.4 Contributions and Significance

This thesis makes the following contributions to interprocedural, causality-aware software vulnerability analysis. Each contribution relates directly to the research objectives and collectively supports the goal of reconstructing executable interprocedural causal chains.

C1 – A chain-centric interprocedural program representation: This thesis extends the Code Property Graph with explicit interprocedural semantics by integrating call links, argument to parameter bindings, return to caller mappings, and conservative alias edges across functions and files. This unified, heterogeneous graph preserves control and data dependencies beyond intra-procedural boundaries and is explicitly designed to support reconstruction of executable root→propagation→sink chains.

C2 - Interprocedural causal vulnerability chain construction: A context-driven causal pipeline that reconstructs explicit interprocedural root-to-sink vulnerability chains over a unified program graph. This approach constructs the full propagation path (source→propagation→sink) as the main explanatory unit, combining structure-aware embeddings (for example, GraphCodeBERT), a heterogeneous graph encoder, and Adaptive Causal Contextualization. Constrained beam search, guided by a compact Causal Knowledge Graph prior, ensures reachable and interprocedurally valid, executable chains consistent with real program behavior.

C3 – Causal evaluation criteria for vulnerability detection: This thesis evaluates the causal vulnerability chain with causal evaluation metrics, such as the Counterfactual Consistency Score (CCS) and the Causal Feature Attribution Measure (CFAM), that assess whether the model's decisions are consistent and based on causal features to ensure the causal faithfulness and robustness.

Significance: This research transforms vulnerability explanation from localized, retrospective feature highlights into a unified, executable interprocedural causal chain that traces the propagation from root cause to exploitable sink. This approach helps bridge the gap between traditional static and dynamic analyses, fuzzing and penetration testing, and modern data-driven vulnerability detectors, aligning automated analysis more closely with the complexity of large, modular software systems. The research makes model predictions interpretable by reconstructing how untrusted input enters the system, travels through control and data dependencies across functions, avoids or skips sanitization, and reaches a vulnerable operation. These chains give developers clear, traceable proof of a vulnerability's existence, as well as clear instructions on where to find it and what to do to fix it. This turns black-box detections into information that can be used and checked. This makes learning-based vulnerability detection more reliable, open, and useful for making sure that software is safe in the real world.

1.5 Thesis Outline

Chapter 1 introduces the problem of interprocedural vulnerability detection, states the research gap and goals, and outlines the proposed approach and contributions. Chapter 2 presents the literature review, covering program analysis foundations, data-driven vulnerability detection, and recent work on explainability and causal

reasoning. Chapter 3 describes the overall methodology, including dataset construction, graph building, and the learning pipeline. Chapter 4 details the evaluation protocol and metrics used to assess the proposed approach. Chapter 5 reports the experimental results and analysis, including robustness studies and qualitative case examples. Chapter 6 concludes the thesis, summarizes the main findings, and discusses future research directions.

Chapter 7 records the core algorithms and pseudocode referenced in the methodology. Chapter 8 provides the formal model architecture, equations, and training objective. Chapter 9 defines the evaluation metrics in detail and presents diagnostic analyses. Chapter 10 lists extended results, ablations, and hyperparameters. Chapter 11 presents the dataset card and licensing information. Chapter 12 documents the role lexicon and pattern rules used to identify sources, sinks, and related roles. Chapter 13 summarizes the reproducibility checklist and artifact details required to replicate the experiments.

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 Theoretical Foundations and Background

This section summarizes key theoretical concepts and technical background for this thesis, establishing the basis for the research and its analytical approach.

2.1.1 Software Bugs and Vulnerabilities

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

This mechanism can be conceptualized as a chain of causation that begins at a *root* condition (e.g., untrusted input, unchecked buffer length, use-after-free) and *propagates* through assignments, parameter passing, return values, aliasing relations, and implicit flows induced by control dependence. The chain becomes exploitable when a tainted or unsafe state reaches a sensitive *sink*, such as a buffer

write, command/SQL execution, deserialization routine, or privileged operation. Executability depends on dominance and post-dominance of guards, exceptional paths, resource lifetimes, and other feasibility constraints. This thesis therefore treats vulnerabilities as *root→propagation→sink* mechanisms to be reconstructed and validated end to end.

2.1.2 Classical Program Analysis: Static and Dynamic

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

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

2.1.3 Code Structures for Security Analysis

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

Control-Flow Graph (CFG): A CFG represents the flow of control between basic blocks, including exceptional control. Analyses derive dominance/post-dominance, loop structure, and reachability—all necessary for judging whether guarding conditions actually protect sinks. Interprocedural CFGs (ICFGs) connect

call sites to callees and returns to callers, modeling control transfer across procedures and enabling reachability checks across functions.

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

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

Code Property Graph (CPG): The CPG combines AST, CFG, and DFG into a heterogeneous, typed multigraph, supporting joint queries over syntax, execution order, and data dependencies [63]. CPGs are well suited to vulnerability discovery because they admit taint-style queries that trace feasible paths from sources through transformations to sinks while retaining syntactic anchors for precise localization. For interprocedural analysis, the CPG is enriched with call-graph edges, argument→parameter and return→caller links, call-site context (e.g., dynamic dispatch), and coarse alias information, enabling faithful reconstruction of cross-boundary propagation. This thesis adopts an augmented CPG as the backbone for chain-centric mechanism tracing.

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

2.1.4 Learning over Program Graphs

Learning-based techniques complement classical analysis by operating directly on structured code graphs. In this view, programs are represented as heterogeneous graphs that merge AST, CFG, and DFG into a unified Code Property Graph (CPG), exposing typed nodes/edges (e.g., `CFG`, `DFG`, `CALL`, `ARG`→`PARAM`, `RET`→`CALLER`). Graph neural networks (GNNs) then propagate information along these relations to classify snapshots or localize vulnerable regions, with statement/line-level variants explicitly exploiting structure to sharpen localization [31, 19, 6]. Relation-aware message passing and attention over edge types are particularly well-suited to interprocedural reasoning because they can weight control-, data-, and call/return links differently during aggregation.

Structure-aware pretraining further improves node/edge representations before

graph reasoning. In particular, GraphCodeBERT injects data-flow awareness into transformer encoders and yields context-sensitive token embeddings that, when fused with structural channels, improve cross-project generalization [17, 13]. In this thesis, such frozen language-model features provide a stable semantic channel, while the relation-aware GNN learns how evidence should travel across heterogeneous edges in the CPG.

Despite these advances, empirical studies consistently show degradation under benign refactorings and cross-repository transfer, revealing sensitivity to spurious, project-specific correlations [48, 64]. This motivates going beyond correlation-based detectors toward mechanisms that privilege executable flows. Recent lines of work introduce causal notions at different levels: contrastive/counterfactual training and inference (*COCA*, *CFExplainer*) [3, 9], causal adjustment and denoising to suppress non-causal context (*Snopy*) [4], and structural-causal formulations that prioritize features with genuine influence (*VulCausal*, *CausalVul*) [24, 44]. Broader explainability studies also highlight the gap between saliency-style rationales and truly causal explanations, calling for evaluations that test faithfulness and intervention stability [36, 27, 28].

These observations motivate the chain-centric stance taken here: use a unified, interprocedural program graph; learn relation-aware evidence propagation over it; and then *constrain* decoding so that the returned explanation is a single, executable root→propagation→sink chain. Causal priors and structured decoding (detailed next) ensure that learned scores translate into mechanisms that remain valid under program semantics and robust under counterfactual edits.

2.1.5 Causal-Guided Structured Decoding

Causal Knowledge Graph(CKG): The concept of a *Causal Knowledge Graph (CKG)* has emerged in recent literature as a means of capturing the underlying mechanisms that explain *why* specific structural or semantic relationships tend to follow one another, rather than merely documenting their co-occurrence [24, 3]. Within program analysis, such knowledge graphs enable models to represent and reason about directional causal dependencies between program relations. Unlike traditional graph structures, which primarily record entities and factual edges, a CKG encodes empirical patterns and causal tendencies, thereby providing a foundation for more interpretable, mechanism-aware reasoning [44, 69].

In the context of vulnerability detection, this thesis leverages CKGs to model causal pathways underlying interprocedural vulnerability propagation [24, 3]. Nodes in the CKG correspond to relation types within heterogeneous program graphs, such as function calls, argument-to-parameter bindings, return-to-caller edges, or data-flow and control-flow links. Directed edges represent empirically observed transitions, indicating where one relation type tends to cause or enable another during the construction of vulnerability chains. Beyond simple one-step transitions,

the CKG summarizes multi-step motifs—recurrent bi-grams and tri-grams—that characterize common propagation mechanisms in actual codebases. For instance, frequent motifs may describe a function call followed by argument propagation and then a return linkage, reflecting a typical cross-functional vulnerability transfer [44, 17].

The value of a CKG lies in its ability to guide learning systems away from brittle, correlation-driven associations. Purely data-driven detectors risk overfitting to locally prevalent regularities that lack mechanistic grounding, resulting in poor robustness under domain shift [48]. A CKG estimated post hoc from training graphs acts as a lightweight structural prior, gently nudging chain assembly toward empirically plausible transition patterns without enforcing rigid rules or handcrafted heuristics [24, 44]. This structure regularizes the inference process, reducing uncertainty and encouraging coherent, interpretable chain construction.

Construction of a CKG relies on analyzing relational evidence from large program datasets to derive statistically frequent transition patterns between relation types. These patterns are extended into higher-order motifs, creating a compact summary of causal sequence structures across interprocedural contexts. The resulting graph reflects both likely cause-effect transitions and interpretable motifs that can be visualized or audited by developers [24, 69]. Importantly, the CKG operates as an auxiliary, non-intrusive source of plausibility: learned evidence from the main detection model remains paramount, with the CKG acting as a gentle regularizer that rewards transitions reflecting historically robust causal structure.

Integration of a CKG into vulnerability detection aligns with ongoing research in explainable artificial intelligence and causal inference, which prioritize mechanistic interpretability over mere statistical association [6, 28]. Prior studies highlight the pitfalls of models lacking causal grounding, noting that visual explanations derived from attention or saliency scores often misrepresent the actual factors driving a prediction [6, 4]. By explicitly encoding directional causal patterns among program relations, the CKG supports stability and transparency in chain reconstruction. Each causal link can be contextualized within a known relational motif, making the resulting causal pathway auditable and actionable for developers [24, 9].

Overall, while traditional knowledge graphs serve mainly as record-keepers for observed facts, a CKG distinguishes itself by focusing on mechanistic tendencies and supporting reasoning under hypothetical interventions, such as what happens when a relation is inserted, altered, or removed. This ability to model executable vulnerability chains with explicit causal grounding positions CKGs as essential tools in mechanistic vulnerability analysis, where understanding the propagation mechanism is as important as detecting the flaw itself [24, 3, 44].

Constrained Decoding and Beam Search for Structured Reasoning: Beam search has become a widely adopted heuristic for navigating complex structured search spaces, enabling the efficient identification of high-quality candidate solutions by

maintaining a finite set of top-scoring partial hypotheses at each expansion step [45]. This approach balances exploration with computational tractability, making it particularly useful in combinatorial prediction tasks. In the context of structured reasoning and program analysis, however, conventional beam search and naive methods such as best-first or depth-first search often struggle with the feasibility constraints imposed by programming language semantics and logical requirements. These unconstrained strategies can produce locally optimal but globally invalid paths, resulting in interpretations or program traces that do not respect the logic of the system.

The concept of *constrained decoding*, also known as guided beam search, addresses this limitation by enforcing domain-specific validity criteria during the search process [38]. In constrained beam search, candidate expansions are dynamically filtered or weighted based on both hard structural constraints (e.g., grammar rules, type safety, execution order) and soft statistical preferences drawn from prior knowledge or data. For program graph reasoning, these constraints typically include control- and data-flow consistency, legality of interprocedural transitions (such as proper function call/return matching), avoidance of cycles or infinite recursion, and aliasing concerns. Incorporating these checks ensures the resulting reasoning chains or traces correspond to paths that are executable and semantically valid, preserving logical coherence across function and module boundaries [9].

Practical applications of constrained beam search span diverse structured prediction problems such as syntactic parsing [12], symbolic program execution [48], knowledge graph inference, and code generation. Earlier methods often relied on hand-crafted heuristics or deterministic rules for pruning search paths, while others introduced scoring functions based on local learned features. However, exclusive dependence on heuristics can fail to adapt across domains, and purely data-driven scoring may lack generalizability and fall prey to spurious correlations. Addressing these shortcomings, recent research integrates statistical priors mined from empirical data, such as edge-gram or motif frequencies directly into the decoding process. These data-driven priors help guide the search toward patterns historically associated with correctness, blending empirical knowledge with strict legality filters [24, 44].

Compared to more global but computationally intensive optimization methods like Integer Linear Programming (ILP) or Satisfiability Modulo Theory (SMT) solving, constrained beam search achieves a practical and interpretable compromise [35]. It enables efficient exploration of numerous plausible causal chains, generating a ranked collection of candidate explanations suitable for developer inspection and verification. The synergy between legal constraint enforcement and empirical causal priors produces a decoding process that is both mechanism-aware and computationally tractable, thereby supporting the broader goal of generating structured, executable explanations that faithfully represent how vulnerabilities propagate in complex software systems.

2.2 Related Works

2.2.1 Static Analysis: Foundations and DL Integration

Static analysis examines software artifacts without executing them, reasoning exhaustively about possible program behaviors through frameworks like lattice-based dataflow analysis and abstract interpretation [63, 6]. Its precision depends on sensitivities to control flow, call context, execution paths, and heap or field representations. Interprocedural static analysis demands accurate call graph construction, supported by methods such as Class Hierarchy Analysis, Rapid Type Analysis, and points-to analyses to resolve aliasing [1, 32].

These analyses utilize canonical program representations: Abstract Syntax Trees (ASTs) encode syntactic structure; Control-Flow Graphs (CFGs) model executable order and branching; Data-Flow Graphs (DFGs) trace variable value definitions and uses; and Program Dependence Graphs (PDGs) unify data and control dependencies for program slicing. The Code Property Graph (CPG) integrates AST, CFG, and DFG into a heterogeneous multigraph that supports global queries essential for interprocedural taint tracking from sources to sinks [63].

Despite early identification benefits, classical static analysis suffers from false positives due to over-approximation in modeling sanitizers, aliasing, and dynamic dispatch [11, 37]. Automated static analyzers such as Flawfinder, RATS, CPPCheck, SpotBugs, and PMD show variability in detection accuracy and false positive rates across languages like C++ and Java [22]. These limitations motivate the development of DL-augmented static analysis tools. For example, IRIS incorporates large language models (LLMs) with static analysis to infer taint specifications and improve vulnerability detection coverage and false discovery rates, surpassing traditional tools like CodeQL [30].

Further, hybrid approaches blend static analysis with ML classifiers to filter and prioritize warnings, addressing false positives and improving scalability [15]. Static analysis remains foundational, providing structured input to DL models, yet its limitations in handling complex, evolving, and domain-specific codebases remain a persistent challenge [1, 6].

2.2.2 Dynamic Analysis: Execution Evidence and DL Integration

Dynamic analysis observes real or symbolic execution of programs with concrete inputs to detect vulnerabilities manifesting during runtime [62, 61]. Techniques include fuzzing, which automates input mutation to explore program paths; sanitizers, which instrument binaries or source to catch memory errors; and symbolic or concolic execution, which employs solvers to systematically explore feasible paths leading to fault states.

Dynamic methods provide high precision witness traces, essential for actionable debugging, but suffer from path explosion and limited environmental coverage,

particularly in large interprocedural contexts [62]. This gap has incentivized hybrid strategies integrating machine learning to guide input generation or prioritize suspicious execution traces, enhancing coverage efficiency [60].

Recent work employs reinforcement learning and deep models to optimize fuzzing or dynamic analysis workflows, combining symbolic constraints and learned heuristics to uncover hard-to-detect vulnerabilities [52]. These advances promise better balancing precision and exploration in complex, modern software systems.

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

2.2.3 Deep Learning Based Techniques

Deep learning is transforming many areas of computer science and has been applied in research projects. In particular, graph neural networks (GNNs) [19], deep learning models have rapidly shown their promise in vulnerability detection, achieving amazing precision in identifying susceptible code patterns [5, 32, 68]. GNNs offer a natural approach for capturing complex software dependencies by modeling code as graphs, using nodes for elements (e.g., variables, functions) and edges for relationships (e.g., data flow, control flow). Hin et al. [19], for instance, showed how well GNNs identified vulnerabilities at the statement level. With the capacity to learn from the structure and semantic information of code, the models were able to spot vulnerabilities suggestive of buffer overflows or injection issues. Although many of these early deep learning models functioned as "black boxes," impairing the knowledge of their decision-making processes [29, 39], even with their precision. Especially in the context of software security, the ability to understand the reason *why* a model finds a piece of code as vulnerable is crucial for developers to implement fixes [53]. This lack of transparency is a major concern. Furthermore, impeding the acceptance of these approaches in practical software development processes is their difficulty in explaining model predictions. The US Government's executive order [53] on the safe, secure, and trustworthy development and use of artificial intelligence emphasizes how much artificial intelligence (AI) is becoming relied upon in important systems, where erroneous or untrustworthy predictions can have severe consequences.

Zhou et al. [31] presented a comprehensive comparative analysis of factors influencing the performance of deep learning (DL)-based vulnerability detection systems. Recognizing the complexity of software vulnerabilities, the authors sought to systematically evaluate how different design choices affect detection efficacy. To this end, they constructed two distinct datasets, capturing both data reliance and control dependencies extracted from programs, encompassing a diverse set of 126 different vulnerability types. Their methodology involved assessing the quantitative impact of several key elements on vulnerability identification performance.

This included employing techniques for handling unbalanced data, incorporating control dependence information within code representations, and experimenting with various neural network architectures. To facilitate their analysis, Zhou et al. built a DL-based vulnerability detection system leveraging Joern, an expanded open-source code parser. The primary contribution of this work lies in its systematic assessment of the quantitative influence of these elements on vulnerability detection efficacy. The study's results offer valuable insights into the design and development of effective deep learning-based vulnerability detection systems. However, it's important to note that this research did not specifically address causal deep learning (CDL) or explicitly incorporate causal reasoning principles. The authors themselves highlighted the need to accommodate a wider range of vulnerability types, enhance the representation of code features, and overcome the limitations of relying solely on control and data dependencies.

Li et al. [48] performed empirical research on deep learning (DL) models for vulnerability detection. On Devign and MSR data, they polled and replicated nine state-of-the-art (SOTA) models. They looked at and examined model capabilities (agreement, variability, performance on several vulnerability categories, and difficulties in addressing particular code aspects). They investigated how model performance responded to project mix and training data size. They identified significant code aspects applied for prediction using model explanation tools. The main contribution of the writers was a thorough investigation of models of deep learning vulnerability detection. For assessing and contrasting several deep learning models for vulnerability identification, this research offers a useful standard. Still, this study paid little attention to causative factors. The writers urged more investigation on code patterns, possible inclusion of causal detection for better generalization, and addressing of the shortcomings of present model interpretation techniques.

Zou et al. [33] tackled this problem with domain adaptation methods and deep learning. To reduce distribution discrepancies between domains, their CD-VulD system learnt cross-domain representations. Learning token embeddings for generalization across tokens, the CD-VulD system transforms software program representations into token sequences. Abstract high-level representations based on those sequences are built using a deep feature model. Minimizing the distribution divergence between the source and destination domains helps to train cross-domain representations using the metric transfer learning framework (MTLF). The key contribution of the authors was a CD-VulD system learning cross-domain representations of source code via domain adaptation and deep learning. Practical vulnerability identification depends on generalizing across several fields, as models trained on one dataset might not work on another. This research did not, however, use causal reasoning; the authors observed constraints in evaluation scope and dependency on particular architectures.

2.2.4 Causal Reasoning and Causal Deep Learning Techniques

Conventional deep learning techniques have shortcomings, including their capacity to detect erroneous correlations and changes in dataset distribution. Stronger and broader methods have emerged from this. VulCausal, developed by Kuang et al. [24], solves what caused what in neural network models. Finding and eliminating bogus links between API functions, user-defined names, and code structure is VulCausal's primary objective. The authors provide a structural cause model to demonstrate how discovering vulnerabilities is affected by user-defined names, API library IDs, and code structure. The study employs covert adjustment in the reasoning stage to eliminate erroneous correlations and minimize the impact of confounders. By modeling and lowering spurious correlations, VulCausal aims to make vulnerability identification more accurate and consistent than conventional deep learning approaches. This is a crucial first step toward exposing vulnerabilities. The writers mostly included a causal perspective to eliminate misleading correlations and provide more accurate and reliable vulnerability detection. When causal inference techniques were developed, they connected with a whole new level and helped us to better understand the factors causing vulnerability in individuals. One significant fresh concept is the application of backdoor correction and causal reasoning approaches. The approach wasn't flawless, the writers noted, though, as it lacked scalability to extremely big codebases, it wasn't ubiquitous across many computer languages, and it couldn't show direct causality (beyond performance benefits). This indicates that in these fields additional research is required. One should also consider the extent of labour causal inference techniques demand and the requirement of well crafted models. Zelikman et al. [67] on self-taught optimizers also indicate that individuals are still striving to create machine learning models that are more efficient and helpful.

Le et al. [47] investigated how well existing deep learning-based detectors could handle vulnerabilities spanning several base units of code (MBUs). Their empirical evaluation of current deep learning-based detectors revealed a clear deficiency in tackling MBU vulnerabilities, usually overstretching accuracy by focusing on individual base units (IBUs). Although it did not specifically include causal reasoning in the detection technique, this research underlined the need of investigating the features and distribution of MBU vulnerabilities as well as the limits of current datasets. Together with a detailed analysis of their incidence and detection accuracy in modern deep learning (DL)-based detectors, the authors gave a description and categorization of MBU vulnerabilities. The key contribution of the authors was the proposal of a framework for the correct integration of MBU vulnerabilities in DL-based detection. The study emphasizes the significance of evaluating the spectrum of vulnerabilities and of studying detection methods. The problem of MBU vulnerabilities is particularly relevant in modern software development, as code usually involves numerous files and modules. Furthermore, improving the understanding of vulnerability types and their frequency is the research by Nong

et al. [41] on authentic vulnerability generation through pattern mining and deep learning and the study by Woo et al. [58] on identifying 1-day vulnerabilities in reused open-source software components.

Cao et al. [4] proposed Snopy, a deep learning (DL)-based technique bridging sample denoising with causal graph learning to enhance vulnerability identification. Snopy expressly included causal reasoning utilizing a Causality-Aware Graph Attention Network (CA-GAT), a Feature Caching Scheme (FCS), and a Causality-Aware Graph Attention Network (CA-GAT) identifying bogus features. Though areas for future research remain, including generalizability to other programming languages and vulnerability types [58] and more sophisticated ways for mitigating spurious correlations in code, this approach represented a major step toward causal vulnerability detection. Initially, deleting vulnerability-irrelevant code elements and constructs using change-based sample denoising and then developing a Causality-Aware Graph Attention Network (CA-GAT) utilizing Feature Caching Scheme (FCS), Snopy learns causal vulnerability traits. The main contribution of the authors was a change-based method guided by vulnerability-fixing commits (VFCs) automatically removing vulnerability-irrelevant code components. A unique addition is made by the combination of sample denoising with causal graph learning; the usage of VFCs offers a moral approach to finding and eliminating pointless code components. Building strong and dependable vulnerability detection systems depends on one being able to differentiate between causal and spurious aspects.

Ganz et al. [16] focused on addressing data quality, model interpretability, robustness, and contextual sensitivity, thereby enhancing the usefulness of machine learning (ML)-driven vulnerability identification. This research particularly applied causal learning approaches to reduce confusing effects and improve detection robustness. They developed datasets by using a novel neural code augmentation technique. They assessed explanation tactics using a novel approach based on dynamic program analysis. They assessed models on their acquired biases using a novel assessment system based on causal learning. The main contribution of the authors was to provide strategies raising the relevance of learning-based vulnerability detection in practical environments. The study underlines the need to attend to pragmatic issues such as data quality and model interpretability. Relevant to the problems of model interpretability and explainability is the research by Suneja et al. [49] on evaluating model signal-awareness by means of prediction-preserving input minimization, together with the study by Yu et al. [66].

Growing interest in causal deep learning (CDL) results from the limits of current approaches, especially in terms of generalization and the capacity to find the underlying causes of vulnerabilities. By means of CausalVul, Rahman et al. [44] directly addressed the lack of resilience and generalization to out-of-distribution (OD) data in deep learning (DL)-based vulnerability detection. Explicitly leveraging do-calculus and the backdoor criterion, this two-stage approach aimed at identifying and eliminating false features using causal learning algorithms. This

paper clearly shows a clear improvement in causal deep learning (CDL) for vulnerability detection, therefore highlighting the possibilities of causal inference to increase model generalization. The correlations between code features and the vulnerability label were shown by the authors using a causal graph. They then trained models that were less dependent on spurious features and more focused on causal features by the use of do-calculus and the backdoor criterion. The main contribution of the authors was to explicitly address false correlations, thereby introducing a fresh method to increase the generalization and resilience of deep learning (DL)-based vulnerability detection. Two important developments are the application of the backdoor criterion and do-calculus. Real-world applications depend on generalizing OOD data since the spread of vulnerabilities may evolve with time.

Chu et al. [9] addressed the lack of explainability in Graph Neural Networks (GNNs) for vulnerability identification. Using counterfactual reasoning, a type of causal inference, CFExplainer aimed to identify minimal changes to the input code graph that would influence the GNN prediction. This improved explainability by focusing on behaviours that alter the outcome and so provides a "what-if" analytical capability. The approach finds a minimum disruption in the code graph by inverting the prediction of the GNN. This provides developers with useful knowledge and guides them on the necessary changes to eliminate a vulnerability. The authors mostly contributed by developing a counterfactual explanation method for GNN-based vulnerability discovery. Counterfactual thinking helps developers find a more reasonable and workable argument. A better basis for understanding the application of counterfactual reasoning in explainable artificial intelligence is provided by the work of Lucic et al. [34].

Islam et al. [20] presented T5-GCN for vulnerability categorization, localization, and root cause identification. Although the "root cause" was sought for, this research did not specifically use causal deep learning (CDL), causal inference, or causal reasoning methods. Conversely, the explainability part of T5-GCN aims to identify the "root cause" of vulnerabilities, therefore indirectly guiding knowledge of the elements generating the vulnerability. The approach uses DeepLift-SHAP attribution values to determine the relevance of tokens in the code, therefore identifying the basic cause. Along with their classification, location, and a brief static description, the writers mostly contributed a method employing explainable methodologies to identify the root cause of a vulnerability. One original method is to combine GCNs and LLMs. Large language models (LLMs) for code analysis are a fast-expanding field of research; the research of Zelikman et al. Furthermore underlined in [67] on self-taught optimizers, the potential of LLMs in this field is relevant for the application of LLMs in code analysis, and vulnerability detection is the effort of Pearce et al. [42] to assess the code contributions' security of GitHub Copilot.

Cao et al. [3] presented Coca, a framework to enhance the causality and robustness of Graph Neural Networks (GNNs)-based vulnerability detection systems.

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

2.2.5 Explainability and Interpretability Techniques

In the first attempt to tackle the explainability issue, scientists aimed to create techniques that would reveal the inner workings of deep learning models, hence enabling more reasonable and reliable predictions. Proposed by Le et al. [29], GAVulExplainer was one of the first attempts to handle this using a model-agnostic approach, that is, one can apply GAVulExplainer to several deep learning models and genetic algorithms to help find important subgraphs causing vulnerabilities. The basic concept is to build a subgraph emphasizing the important elements causing the vulnerability, therefore offering a more understandable justification for the predictions of the model. While avoiding local optima, the genetic algorithms effectively seek accurate substructure information. This method employs a fidelity metric to assess the quality of produced explanations and lets users regulate the size of the explanation subgraph. GAVulExplainer marks a significant progress in enhancing the interpretability of GNN-based vulnerability detection, but it still lacks explicit modeling of the fundamental *causal* links between code characteristics and vulnerabilities. Finding contributing subgraphs still takes front stage without exploring causal deep learning (CDL) methods. Furthermore, its assessment depends on fidelity criteria, which might not completely reflect the pragmatic value of the explanations for developers in real-world debugging situations, especially given the complexity of software systems and the several ways vulnerabilities might show themselves. This study addressed the demand for explainable prediction since they realized that successful remedial action and developer confidence depend on knowing the "why" behind the prediction of a model.

Li et al. [29] presented that Vulanalyzer is another significant initiative aimed at improving explainability. Specifically developed for binary detection, a rather difficult subject given the low-level features of binary code, this deep learning model, Vulanalyzer, accurately preserves instruction semantics and structural linkages by

using sequential and topological learning to mimic program execution through recurrent units and graph convolution, especially in assembly code [51]. Mostly distinguished by its multi-head attention system, which stresses pertinent commands and fundamental blocks, Vulanalyzer underlines fundamental directions and basic blocks. This encourages developers to concentrate on the code components the model considers most indicative of a vulnerability, therefore producing interpretable results. Although it clarifies things, Vulanalyzer, like GAVulExplainer, does not especially mix causal reasoning, causal inference, or causal deep learning (CDL). Emphasizing the need of greater study on causal approaches that can identify the why behind vulnerability projections, the emphasis stayed on simplifying difficult interactions and improving interpretability by means of attention processes. The writers mostly concentrated on the lack of explainability and the challenges to elucidate intricate links in binary code. With topological and sequential learning combined, the approach captures structural relationships as well as instructional meanings, so requiring minimal topic knowledge. Still, even if attention processes help to define the what of the model's judgments, they do not naturally define the *why* - the fundamental causal factors generating the sensitivity. Most importantly, the method can be applied over multiple architectures and code obfuscation methods. The research on the development of explainable functional summaries of assembly code performed by Taviss et al. [51] emphasizes understanding code semantics in vulnerability analysis.

Moschitti et al. [28] presented an XAI-based system for assessing computer code in a graph environment. This system evaluates the significance of syntactic structures for Common Weakness Enumeration (CWE) classification [27], therefore connecting learnt code feature representations to subtle semantics recognized by security professionals. The system generates ranks of syntactic constructive contribution levels in Abstract Syntactic Trees (AST) among CWE types for Java and C++ datasets. A novel feature-masking method, varying the neighbourhood of code tokens and syntactic constructions, is applied for the graph environment. The change in the code token neighbourhood is transformed into the CWE-type similarity score by means of information retrieval approaches. The authors showed how nuanced semantics understood by security professionals might be connected to the learnt code feature representations by CWE similarity generated from XAI explanations. This approach did not, however, particularly target causal reasoning or causal deep learning (CDL). The authors admitted that present XAI systems have several limits, namely, their incapacity to generalize to undiscovered vulnerability patterns and transcend the scope of input data. The constraints covered are those of interpretability of acquired features and transferability of learned patterns to different datasets. Although graph-based representations and ASTs are widely used in vulnerability identification, their efficacy may vary depending on the programming language and the particular vulnerabilities under attack. The research of Allamanis et al. [1] on machine learning for massive code and naturalness offers a larger background

for appreciating the difficulties of expressing and evaluating code.

Hajipour et al. [25] developed a framework for vulnerability threat prediction. This method generated a semantic representation and computed an explainable threat score by prioritizing research activities using topic modeling of vulnerability descriptions (from sources like the National Vulnerability Database). Furthermore, included was a fresh trend score based on internet infosec conversations to pinpoint popular vulnerabilities. These results were aggregated on a visual dashboard to give investigative work top priority. The main contribution of the authors was the computation of a new trend score and an explainable threat score based on the topic model. This framework offers a semantic representation of vulnerabilities constructed using topic modeling of vulnerability descriptions. The framework offers a semantic representation of vulnerabilities constructed using topic modeling of vulnerability descriptions. Although helpful for prioritizing, it did not investigate the fundamental *causal* elements affecting vulnerability exploitability, like particular code patterns or interactions increasing the probability of the exploitation of a vulnerability. Understanding which vulnerabilities are most likely to be exploited aids in prioritizing risk management efforts. Existing frameworks often rely on online discussions and external vulnerability descriptions, which may introduce biases and limit timeliness and completeness.

Allix et al. [36] investigated the use of deep learning and explainability methods (most especially SHAP) toward this aim. Their results showed that explainability techniques might occasionally present distracting information, therefore harming rather than supporting engineers; code characteristics employed by DL models were often only partially connected to the underlying causes of vulnerabilities. This emphasized the need for more accurate localization and the incorporation of causal reasoning to uncover the true underlying causes. This allows for a deeper understanding that goes beyond mere correlations. The main contribution of the authors was to assess how well explainability and deep learning (DL) approaches could localize source code assertions concerning vulnerabilities. Using two deep learning (DL) techniques, VulDeePecker and JavaBert, the authors localized source code phrases pertaining to vulnerabilities using SHAP, a model-agnostic explainability method. The paper emphasizes that current explainability methods do not provide developers with practical insights effectively. Important determinants of the results of the study are the choice of deep learning models (VulDeePecker and JavaBert) and the respective application of SHAP. Practical implementations depend critically on more programming language-oriented encoding approaches recommended by Allix et al. [36] and the evolution of more advanced techniques for vulnerability localization.

2.3 Research Gaps

Automatic vulnerability detection has advanced considerably, yet existing systems remain limited in reliability, generalization, and practical applicability. These limitations motivate the chain-centric, interprocedural perspective adopted in this thesis.

Gap 1: Causal deficiency and generalization: Many data-driven detectors learn superficial statistical correlations rather than the underlying flaw mechanisms. Empirical studies show that their predictions often degrade under harmless refactorings, renaming, formatting changes, or cross-project transfer, indicating a lack of causal grounding and poor out-of-distribution robustness [48, 64].

Gap 2: Interprocedural reasoning and chain reconstruction: Most existing approaches operate at the level of individual functions or narrow intra-procedural slices, capturing only limited contextual information. They generally fail to establish explicit interprocedural links connecting root causes, propagation paths, and final sinks [47]. Consequently, these methods cannot reconstruct executable root→propagation→sink chains that accurately trace vulnerable states across large, modular software systems.

Gap 3: Chain-centric evaluation and causal faithfulness: Standard metrics such as Accuracy, Precision, Recall, and F1-score measure label agreement but say little about whether explanations are causally faithful or not. Existing causal and explainability frameworks offer localized insights but lack widely adopted chain-centric metrics for assessing alignment with executable interprocedural behavior and counterfactual responses [4, 24, 44, 9].

A more effective approach must explicitly model interprocedural control and data dependencies, reconstruct executable source-to-sink chains, and distinguish causal features from spurious correlations. This thesis addresses these gaps by designing an interprocedural program representation and a context-driven causal reasoning process that together aim to reconstruct and evaluate executable interprocedural vulnerability propagation chains.

2.4 Research Challenges and Rationale

Automated vulnerability detection has advanced, but reliability, generalization, and interpretability remain limited. Public corpora often suffer from duplicated entries, weak or noisy labels, and severe class imbalance, which can inflate reported performance and hinder reproducibility [48, 6]. Many datasets expose only function-level snippets and omit cross-function flows, making interprocedural reasoning difficult. *ReposVul* improves curation by preserving repository context and call relationships, but it has not yet been fully exploited for chain-centric analysis [55].

Representation is a central challenge. Sequence models perform well on benchmarks but largely ignore control and data dependencies, leaving them vulnerable to simple perturbations [6, 13]. Graph-based views (AST, CFG, DFG, PDG, CPG) better capture structure [31], yet practical performance depends on how precisely interprocedural details such as calls, argument→parameter links, returns, and aliasing are modeled. Structure-aware pretraining, exemplified by GraphCodeBERT, injects data-flow signals into token embeddings [17], but its benefit ultimately depends on the quality and coverage of the underlying program graph.

Constructing executable root-to-sink chains poses an additional difficulty. Unconstrained path search tends to drift, while global solvers are computationally expensive. This thesis instead uses constrained beam search coupled with Adaptive Causal Contextualization (ACC): the decoder admits only steps that satisfy reachability, call/return discipline, and alias checks, while a compact Causal Knowledge Graph (CKG) prior, mined from relation n-grams, guides transitions without sacrificing semantic validity.

Model fragility across projects, time, and refactorings remains a persistent issue [48]. Semantics-preserving edits can flip predictions, and post-hoc explanations often emphasize misleading cues [36, 28]. Recent causal approaches based on counterfactual training and graph edits improve robustness but lack standardized metrics for causal fidelity [3, 9, 44, 24, 4].

This thesis responds to these challenges by (i) representing programs as enriched, interprocedural CPGs with explicit cross-boundary semantics, (ii) decoding a single executable chain from root to sink under structural and causal constraints, and (iii) evaluating explanations with both standard detection metrics and causal tests, including counterfactual consistency, directional agreement, on-chain attribution, and chain invalidation.

Chapter 3

Methodology

This chapter shows the learning pipeline that was used in this thesis. The objectives are twofold: precise identification of vulnerabilities and accurate reconstruction of executable interprocedural chains. The method connects a root cause to its spread and its end point through paths that don't interfere with the program's execution. This talks about the problems with models that draw lines in isolation and don't take into account how they work.

I work with an enhanced program graph. I use a Code Property Graph to show programs. This is a joint, labelled graph that combines the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG) into one searchable structure [63]. I add interprocedural links to the base CPG, such as CALL, ARG→PARAM, and RET→CALLER/RET→LHS, as well as alias summaries. This addition keeps the execution semantics the same across function boundaries and makes it possible to slice precisely for chain reconstruction.

There are three steps in the pipeline. To start, feature initialization uses GraphCodeBERT to encode tokens with context that knows about structure and data flow [17]. Pretraining on a large scale gives you useful information that you can use for later learning. Secondly, a graph attention encoder collects signals from different parts of the graph that are close to each other. Executable relations like def–use and control dependencies guide attention. Regularization that focuses on causality lowers the need for false correlations and makes generalization better [3, 44, 24]. Third, Adaptive Causal Contextualization (ACC) puts together an executable chain by choosing paths that are limited. It makes sure that data and control are possible, follows the call-return structure, and prefers short explanations. The end result is a single chain that can be understood and matches real execution.

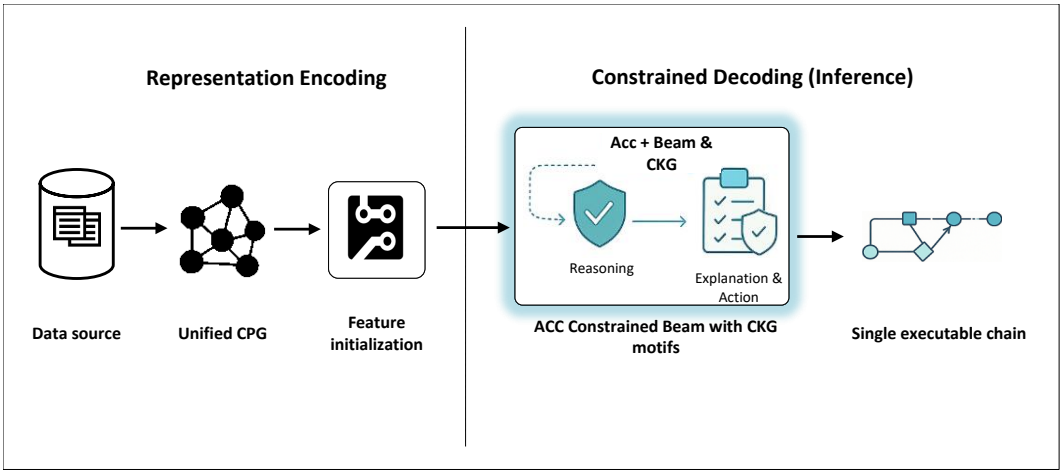

Figure 3.1: End-to-end pipeline for chain-centric vulnerability detection. Source repositories are parsed into a unified Code Property Graph (CPG), with GraphCodeBERT-initialized features providing input to a relation-aware encoder. At inference, ACC-constrained beam search guided by CKG motifs performs structured reasoning and outputs a single executable interprocedural chain, enabling interpretable vulnerability mechanisms.

Training includes a classification loss for detection and penalties that make it less likely that attention patterns will be non-executable or incoherent. The learned representations maintain stability during benign refactorings and facilitate transfer across projects and programming languages. The design is based on new developments in structure-aware vulnerability detection and learning to represent causes [31, 48, 4, 9, 19].

The rest of the chapter talks about the formal setting, the optimization, and the algorithms for each step.

3.1 Chain-Centric Program Representation

3.1.1 Dataset Description, Ground Truth Formation, and Preprocessing Pipeline

This section specifies the experimental corpus, the ground-truth labelling protocol, and the preprocessing that transforms repositories into chain-centric interprocedural graphs. I utilize *ReposVul*, a repository-level dataset that separates patches, maintains multi-granularity dependencies, and eliminates outdated fixes [55].

3.1.2 ReposVul: scope and characteristics

ReposVul links CVE/CWE records to pre- and post-fix code snapshots, patch commits, and repository context [55]. Each entry pairs an identified weakness and severity with the exact commit, message, files, and diffs that implement the fix. Corpus construction crawls public sources, untangles mixed commits to keep only fix-relevant files, mines cross-file caller–callee relations, and filters superseded patches by commit history. The result is a repository-level view that preserves chronology, captures interprocedural links, and supports chain-centric evaluation.

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

3.1.3 Extraction and graph construction pipeline

The conversion from raw dataset entries to chain-centric program graphs occurs over six sequential stages.

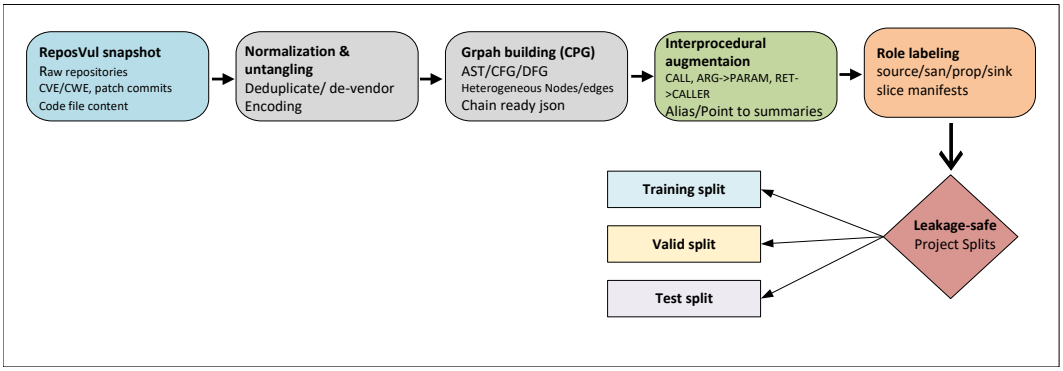

Figure 3.2: ReposVul prepossessing to interprocedural CPGs and leakage-safe train/valid/test partitions.

Stage A: Getting raw vulnerability and patch information: I add CVE/CWE metadata to *ReposVul* entries and get the repository at parent and child commits.

This keeps all of the pre- and post-fix files. For traceability, commit IDs, messages, dates, project IDs, and file paths are all indexed.

Stage B: figuring out the vulnerabilities: Patches often combine fixes with changes to the code. I use the corpus untangling rule, which combines model judgments with static cues, to keep only files that are relevant to fixing.

Stage C: extraction of dependencies at multiple levels of granularity: For retained patches, I get links between callers and callees from all over the repository. When necessary, the scope includes top-level functions so that interprocedural paths from candidate sources to sinks can be recorded.

Stage D: making the code more consistent and building the static graph: For each snapshot, I make a heterogeneous program graph that combines AST, CFG, DFG, call edges, ARG→PARAM, and RET→CALLER links. Pointer and container def-use are similar to conservative alias relations. Identifiers are made anonymous, and literals are put into buckets to cut down on the differences between harmless edits [6, 48].

Stage E: differencing that takes patches into account and slice materialization: I calculate line diffs for each patch and then create four synchronized views: changed lines, enclosing functions, touched files, and a repository subgraph that can reach any sink through possible control and data flow. These views are what you use to put together a chain.

Stage F: filtering based on traces for old patches: I keep track of file paths and commit times so I can get rid of files that don't work and patches that are no longer needed.

Table 3.2: Preparation pipeline summary.

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

3.1.4 Unified multigraph, typing, features, and storage

I depict each repository snapshot as a heterogeneous multigraph constructed upon the Code Property Graph (CPG) abstraction [63]. A single typed node store keeps program parts. Relation-specific edge sets encode AST, CFG, and DFG. Interprocedural links are CALL (caller to callee entry), ARG2PARAM (actual to formal), RET2CALL (callee return to call site), and RET2LHS (return value to assignment target). For all families, reverse edges are made real. This topology shows the control and data dependencies and cross-boundary flows that are needed to rebuild the chain.

For example, an identifier, literal, operator, statement, basic block, or function can be used to type nodes. Each node has a small structural feature vector that includes flags for token categories, simplified SSA indices, type hints, and normalization markers. Edges are classified by family and orientation. Optional fields include guard predicates on CFG edges and alias provenance on links that come from points-to. Shard files hold graphs, and each graph has its own metadata, such as the repository, commit, file path, and function names. This makes it possible to store things in a way that can grow and look up their origins quickly.

I keep two encodings going at the same time. The *base* encoding has only small structural features for ablations and classical GNNs. The "pretrained" encoding adds a 768-dimensional embedding from GraphCodeBERT [17] to each node. The outcome is a dual-channel configuration: a low-dimensional structural vector combined with a high-dimensional contextual vector. The topology and interprocedural counts are the same for all encodings; only the feature space is different.

The set of relation types is treated as a fixed alphabet $\mathcal{R}$ by decoding. This

includes AST, CFG, DFG, CALL, ARG2PARAM, RET2CALL, RET2LHS, and DFG_-THIN. It changes the weights of acceptable expansions during decoding, but it never changes the graph.

3.1.5 Ground-truth definition

Ground truth is defined at two levels. At the label level, a repository revision is marked vulnerable if it appears in a vulnerability fix pair in ReposVul, and its corresponding fixed revision is labeled non-vulnerable [55]. Using paired revisions with shared context minimizes the risk of unrelated edits affecting label validity

Each positive example is annotated with one executable chain from a source to a sink, possibly crossing function boundaries. Sources mark points where untrusted data enters program state, sanitizers check or constrain that state (for example, bounds or format checks), propagators move taint through assignments, parameter passing, returns, pointer hops, container writes, or index arithmetic, and sinks are security-relevant operations such as memory writes, command execution, path traversal, database calls, or unsafe casts. The full role lexicon and patterns are given in Appendix F.

Chains are constructed by seeding candidates from diffs, following def–use links to identify propagators and sinks, and wiring calls and returns along the repository call graph with ARG → PARAM, RET → CALLER, and RET → LHS bindings. Retained examples satisfy basic consistency conditions (every source reaches a sink, every sanitizer blocks at least one tainted path, and every propagator lies on a CFG-consistent route). A feasibility pass then checks that removing the sink reduces exploitability, and a counterfactual pass strengthens guards or replaces dangerous sinks with safe ones to confirm that reachability changes as expected; failing cases are corrected or discarded.

3.1.6 Interprocedural Semantics and Cross-Boundary Validity

Interprocedural structure is explicitly modeled through a repository-level call graph integrated into each program graph. It represents call contexts, parameter counts, and links between real arguments and formal parameters (ARG→PARAM), as well as return flows (RET→CALLER/LHS). Alias edges capture pointer, reference, and container flows, with each interprocedural edge annotated by its source file and enclosing function.

Repository-level dependency extraction recovers cross-file root-to-sink chains beyond syntactic diffs. Table 3.8 shows that many instances contain reachable caller–callee pairs, confirming cross-function traversability. Feasibility and counterfactual tests along these paths validate that the reconstructed chains are executable and causally coherent.

3.1.7 Data Splits, Controls, and Reproducibility

ReposVul provides official project-level train, validation, and test splits designed to measure cross-project generalization and avoid data contamination from code duplication or patch ancestry. This thesis adopts these splits without modification to preserve benchmark integrity [55]. All files from a given project remain in a single partition, and when temporal robustness is evaluated, the splits respect commit chronology. Parent and child patch commits are never separated, identical file snapshots and CVE instances do not appear in multiple splits, and cross-repository artifacts are not merged across partitions. For reproducibility, preprocessing configurations, graph statistics, split checksums, and relevant metadata are recorded and made available.

3.1.8 Empirical Coverage and Dataset Statistics

Tables 3.6–3.8 report split-wise statistics for the prepared C and C++ subset used in my experiments, including label balance, graph-instance density, and interprocedural connectivity.

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

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

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

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

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

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

These figures show that nearly one third of instances expose a callee set, about one eighth a caller set, and roughly one in ten both, providing the minimal structure needed to discover chains that cross function boundaries. Validated by feasibility and counterfactual checks, this supports true interprocedural vulnerability analysis over isolated intra-procedural signatures.

3.1.9 Splits, controls, and coverage

I use the official, project-disjoint `ReposVul` splits without modification [55]. Each project stays in a single partition. Parent–child patch pairs are not split. Identical file snapshots and duplicate CVEs do not cross partitions. Cross-repository call-graph artifacts are not merged. When temporal robustness is assessed, commit chronology is respected. Reproducibility controls are strict. Each example stores the ordered node IDs of the chain, node roles, edge types, and synchronized line, function, and file indices.

Table 3.6: Split-wise label counts (file-snapshot granularity).

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

Table 3.7: 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}

Table 3.8: Interprocedural connectivity: presence of non-empty caller and callee sets.

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

In almost one-third of cases, a callee set is shown; in about one-eighth of cases, a caller set is shown; and in about one-tenth of cases, both are shown. These rates create the structural conditions that make cross-function chains possible. Since feasibility and counterfactual checks work along these links, kept examples support executable, interprocedural mechanisms instead of isolated intra-procedural cues.

3.1.10 Summary

This approach makes program graphs that are full of procedures and ready to be linked together. Each part has a clear job (source, sanitizer, propagator, sink). Instead of separate statements, these graphs encode executable paths. *ReposVul* gives you reliable label provenance and a wide repository context [55]. Using its official project-disjoint splits, along with normalization and verification, lowers the risks to validity that are already known in learning-based vulnerability detection [6, 48].

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 centralized in Appendix B. See figure 3.3 for a single-page overview that this section follows.

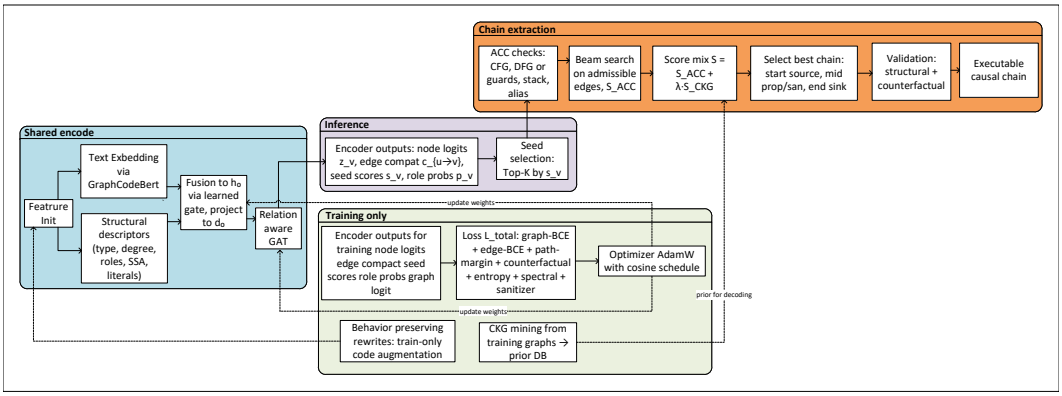

Figure 3.3: Chain-Centric Model Architecture and Inference Flow

I set up node features using code spans and structural descriptors, encode them with a relation-aware GAT, and then decode a single executable chain while checking for ACC feasibility. Inference starts beams from seeds chosen by the model and can also mix in a weak CKG prior for ranking only. Training finds the best composite loss and gets the CKG only from the training split. Solid arrows show how data or scores move, dashed arrows show operations that are only for training, and dotted arrows show the weak decoding prior (CKG) or optimizer feedback to parts of the encoder that can be trained. With the overall attack flow and vulnerability propagation pathways in view, I now proceed to initialize the GraphCodeBERT feature extractor leveraging its pre-trained structural and semantic code understanding to encode interprocedural dependencies and vulnerability relevant patterns into a unified embedding space.

3.2.1 GraphCodeBERT Feature Initialization

Each node that corresponds to a concrete source span is tokenized with GraphCodeBERT's BPE into sliding windows (length L, stride S); per-window contextual token embeddings are then aggregated back to the node. When a fragment contains identifiers, a light edge-aware attention emphasizes tokens that carry def–use signal (Appendix Eqs. B.2–B.3); otherwise a simple average is used (Appendix Eq. B.1). The resulting textual vector $\mathbf{x}_i^{\text{text}}$ is fused with a structural descriptor $\mathbf{x}_i^{\text{struct}}$ (node type, degrees, SSA hints, literal statistics, role flags) by projecting both branches to d_0 and combining them through a learned gate (Appendix Eqs. B.4–B.6), yielding the initialization $\mathbf{h}_i^{(0)}$. If a span appears in multiple windows, pooling is applied per window and then averaged; function-proxy nodes pool child statements to obtain a coarse function representation. When a node and its def–use neighbors fit in one window, GraphCodeBERT's data-flow mask is enabled so these pairs can attend directly. To attribute gains primarily to the graph encoder, GraphCodeBERT remains

frozen by default. Table 3.9 summarizes feature dimensions.

Table 3.9: Node feature dimensions at initialization.

Component	Dim.	Notes
Textual embedding	768	GraphCodeBERT contextual vector [17]
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

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 $c^{(r)}_{u \to v}$ (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-level BCE (class-balanced) with an edge participation BCE and a path-margin term that forces true chains to outrank length-matched admissible random walks; when multiple chains exist, losses aggregate so the encoder learns a distribution over plausible roots.

3.2.3 Causal Knowledge Graph (CKG): Mining and Prior

From training chains only, unigram and bigram statistics (plus a small set of named trigrams for interpretability) 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 (Appendix Eq. B.13) with weight λ. This preserves data-driven evidence while 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 (realizable interprocedural CFG path; justified non-CFG hops; well-nested call stack; alias-consistent pointer hops) and counterfactually (neutralizing the sink, strengthening a dominating sanitizer, or unlinking a taint-carrying call should remove or reroute the chain).

3.2.6 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 used only during decoding, not training. Models train with AdamW, cosine learning-rate decay, two-epoch warmup, gradient clipping, and mixed precision. Class imbalance is handled by weighted sampling. Semantic-preserving augmentations like renaming and code reordering apply with 0.3 probability. Fixed seeds, logged checkpoints, and cached features support reproducibility, with training curves and outcomes archived.

Summary

This section summarizes the framework for reconstructing executable root-to-sink chains in five steps: node encoding with GraphCodeBERT embeddings (Appendix Eqs. B.2–B.6), graph reasoning via a relation-aware GAT (Appendix Eqs. B.7–B.8), constrained decoding guided by a weak Causal Knowledge Graph prior (Appendix Eq. B.13), chain selection through Adaptive Causal Contextualization (ACC) with validation (Appendix Eqs. B.14–B.31), and training using graph-level binary cross-entropy with light regularization (Appendix Eq. B.32). ACC enforces control-flow, data-flow, and aliasing constraints, ensuring that only semantically valid chains are selected.

Chapter 4

Evaluation Protocol and Metrics

This chapter explains how the method is evaluated. Two aspects are considered: (i) conventional classification quality, and (ii) semantic correctness, i.e., whether predicted evidence forms executable interprocedural chains aligned with program semantics.

4.1 Experimental Settings

Here I summarizes datasets, inputs, model variants, decoding methods, and reporting conventions. It then covers thresholding and calibration, conventional metrics, chain focused checks, and counterfactual metrics in all experiments.

Dataset and splits: The official project-disjoint splits of `ReposVul` [55] are used. Projects never cross splits, positive and negative pairs stay within a split, and, when reported, temporal order is preserved.
Inputs: Models work with chain-prepared interprocedural CPGs (AST, CFG, DFG) that have the relations `CALL`, `ARG`→`PARAM`, and `RET`→`CALLER`/`RET`→`LHS` and alias summaries.
Model variants: Two configurations are compared under the same training/decoding regime: *Struct-only* and *GCBERT+Struct* (GraphCodeBERT fused with the same structural descriptors) [17].
Decoding: Inference uses beam search constrained by ACC, with a decoding-only CKG prior mined from training data. The score combination is given in Appendix. B.5. Numeric settings and software versions appear in Appendix. D.1.
Protocol: Early stopping is applied on validation Macro-F1. Five random seeds are used per configuration. Reproduction scripts and configuration files are included in the artifact (Section G).

4.2 Thresholding and Calibration

Two operating points are reported: (i) the validation F1-optimal threshold $\tau_{\mathrm{F1}^\star}$, and (ii) a fixed τ=0.5 for like-for-like comparison. If needed, temperature scaling is fitted on validation and kept fixed at test time.
Expected Calibration Error (ECE):

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

B is the number of confidence bins, S_b is the set of examples in bin b, $|S_b|$ is its size, N is the total number of examples, $\mathrm{acc}(S_b)$ is empirical accuracy in bin b, and $\mathrm{conf}(S_b)$ is the average predicted confidence in that bin. Lower ECE indicates better calibration. Binning choices and variants are in Appendix. C.1.2.

4.3 Standard Classification Metrics

After thresholding, the standard counts TP, FP, TN, and FN are computed. The report includes Precision, Recall, and F1:

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}, \quad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}, \quad \mathrm{F1} = \frac{2\,\mathrm{Precision}\cdot\mathrm{Recall}}{\mathrm{Precision}+\mathrm{Recall}}. \tag{4.2}$$

Precision measures correctness among predicted positives. Recall measures coverage of true positives. F1 balances the two. Accuracy, Macro-/Micro-F1, AUROC, and AUPRC on raw scores are also reported. Because the positive class is rare, AUPRC and Macro-F1 are emphasized. Formal variants are in Appendix. C.2.

4.4 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.3.1.
Structural fidelity: When a reference chain is available, reported metrics include node and edge coverage, longest common subsequence ratio, and role-aware coverage for source, sanitizer, propagator, sink. Formulas are detailed in Appendix C.3.2.
Interprocedurality: The IPA Rate quantifies the share of predicted chains that traverse `CALL`/`ARG`$\rightarrow$`PARAM`/`RET` edges when such edges exist in the slice (Appendix. C.3.3).

4.5 Counterfactual Metrics

I evaluate causal robustness via three targeted edits - guard strengthening, call unbinding, and sink neutralization. Results are reported by intervention type with confidence intervals and methods detailed in Appendix. C.4.1 and Appendix. C.4.2. The CKG prior is active except during robustness tests when it is disabled on the edited graph.

Counterfactual Consistency Score (CCS):

$$\mathrm{CCS}_i = (p_i - p_i^{\mathrm{do}})^2. \tag{4.3}$$

For the graph Before the edit, p_i is the predicted probability, and after the edit, p_i^{do} is the new probability (both are between 0 and 1). A small value means not much has changed, while a large value means a big change has happened. In Appendix. C.4.1, a directional consistency rate that checks to see if the change is going in the right direction for each type of edit is defined.

Causal Feature Attribution Measure (CFAM):

$$\mathrm{CFAM}_i = \frac{\sum_{f \in F_c} A_i(f)}{\sum_{f \in F_c \cup F_s} A_i(f)} \in [0, 1]. \tag{4.4}$$

F_c and F_s are sets of features that are both on-chain and off-chain. The attribution given to feature f for graph i is $A_i(f) \geq 0$. The numerator adds up the attribution for features that are on the chain that was returned. The denominator adds up the attribution for all the features. Values that are closer to 1 mean that the features on the returned chain support the choices. Normalization choices can be found in Appendix. C.4.2.

4.6 Decoding Diagnostics

There are four indicators that sum up decoding dynamics: the Chain Success Rate, the average chain length, the interprocedural hop ratio, and the ACC rejection mix. Appendix. C.5 and the supplement show more diagnostics, such as the admissible expansion ratio, motif coverage, and prior influence rate. The dataset includes graph predictions, chains with beam traces, calibration parameters, intervention logs, configurations, seeds, environment hashes, the CKG prior, and decoding diagnostics. Reproducibility scripts enable regenerating graphs from raw repositories and recreating all reported tables and figures. Full details and instructions are in Appendix G.

Chapter 5

Experimental Results and Analysis

This chapter evaluates the chain-centric methodology for interprocedural vulnerability detection, assessing causal validity, efficiency, and reproducibility. All results are structured to support incremental extension while preserving methodological coherence.

5.1 Experimental Setup and Configuration

Experiments use the official `ReposVul` project-disjoint splits. Commit chronology is respected. Two encoders are compared: *Struct-only* and *GCBERT+Struct*. The language model remains frozen. Decoding is ACC-constrained beam search. A decoding-only CKG prior is applied.

Table 5.1: Experimental setup summary.

Item	**Setting**
Dataset	`ReposVul`, project-disjoint, chronology respected
Graph relations	DFG, CFG, CALL, ARG2PARAM, RET2CALL, RET2LHS; alias summaries on
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 5.2: Dataset split summary.

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

Model variants: The model variants include Struct-only, which uses compact structural descriptors such as types, degrees, SSA hints, and literal buckets. The GCBERT+Struct variant combines GraphCodeBERT embeddings with the same structural features. Both variants share identical topology, and the language model remains frozen unless otherwise specified.

Table 5.3: Struct-only vs. GCBERT on the same shard.

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

Decoding uses ACC-constrained beam search, allowing steps only if CFG reachability holds, with ARG to PARAM and RET to CALLER/LHS bindings consistent, alias checks passing, and stack discipline maintained. The CKG prior, mined from training graphs, only influences move weights at inference.

Table 5.4: Decoding, training, and evaluation protocol.

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	GCBERT 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

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.

5.2 Conventional Metrics

For both encoder variants, In this section I report standard graph-level metrics like Accuracy, Precision, Recall, F1, AUROC, and AUPRC. I set the operating threshold $\tau_{F1^\star}$ on the validation split and leave it the same on TEST. I use temperature scaling to adjust probabilities on validation and then use the same temperature on TEST. I use AUPRC as the main score and F1 and AUROC as context because TEST is unbalanced (positives $\approx 2.9\%$). I average the results over five seeds.

Table 5.5: Validation and test metrics at $\tau_{F1^\star}$ (mean over five seeds).

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

GCBERT+Struct improves AUPRC and F1 on both splits (AUPRC +0.15, F1 +0.14 on VALID; AUPRC +0.15, F1 +0.13 on TEST). Accuracy is high for both due to class imbalance. AUROC indicates stable ranking.

Table 5.6: Operating points and calibration. ECE uses 15 bins.

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	*from Valid*	0.082	**0.036**
Test	GCBERT+Struct	0.27	*from Valid*	0.064	**0.022**

Calibration improves ECE on both variants after scaling. PR and ROC curves (artifact) reflect the same ordering.

Table 5.7: Confusion matrices on TEST at $\tau_{F1^\star}$.

	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

Error patterns are consistent: false positives frequently stem from call-heavy wrappers with weak guards or format string construction logic; false negatives typically arise in macro-expanded code or callback-driven paths.

5.3 Executable Chain Quality and Interprocedural Evidence

Chains are evaluated as executable sequences with ACC-constrained beams set to $K = 8$, $B = 24$, $H = 5$, and $\alpha = 0.7$. A decoding-only CKG prior with $\lambda = 0.2$ weights allowed moves. Statistics are calculated on positive slices yielding chains, with results averaged over five seeds. A chain is feasible only when interprocedural control flow is realizable, data flow or guard preservation holds, call/return are matched, and alias checks pass. The GraphCodeBERT variant increases feasibility by 8–9 points on both splits.

Table 5.8: 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

The IPA rate, the share using both call and return, and the mean call depth are all reported for cross-boundary edges (CALL, ARG→PARAM, and RET→CALLER/LHS).

Table 5.9: 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**

Node and edge coverage, role coverage (source, sanitizer, propagator, sink), and

order agreement (LCS) are reported. Sanitizer recovery and edge coverage have the biggest gains. The order agreement also gets better.

Table 5.10: 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 5.11: 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

The hop count, the number of files crossed, and the percentage of summary edges are all given. Chains stay small and only cross a few files. Relying on summary edges lessens while agreement rises.

Table 5.12: Chain length and span: mean/median hops, files crossed, and share of summary edges.

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**

In summary, the *GCBERT+Struct* variant is more feasible, has better interprocedural use, better role and edge coverage, a higher LCS, and a chain length that is about the same.

5.4 Causal Faithfulness

I assess prediction dependence on reconstructed chains through minor and major counterfactual modifications. The *Counterfactual Consistency Score* (CCS) measures

the magnitude of probability change. The *Directional Consistency Rate* (DCR) indicates the proportion of edits causing sign changes. The *Causal Feature Attribution Measure* (CFAM) quantifies attribution to the predicted chain. The Chain Invalidation tells how many cases there are where no valid chain is left after editing.

Table 5.13: 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

Small CCS shows up after small changes. When the sink is neutralized, a larger CCS shows up supporting the hypothesis that the chain endpoint drives the prediction.

Table 5.14: CCS by edit type (Lower is better for minor edits; 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**

Direction agrees with the intervention in most cases. The GCBERT+Struct variant is consistently higher by (0.03!-!0.04) absolute across splits.

Table 5.15: 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**

Attribution fits with the expected chain. The enriched version raises the on-chain share, with the biggest gains for propagators and sanitizers.

Table 5.16: 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

Editing the sink removes almost all admissible chains. Minor edits invalidate about half of them. Score drops are large when a chain survives a sink edit.

Table 5.17: 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

The evidence is consistent across splits. Minor changes lead to small CCS and high DCR. Major changes result in large CCS, higher DCR, and near-complete chain invalidation. CFAM increases with the enriched encoder, focusing attribution on key segments. These results support the idea that predictions depend on executable chains instead of random context.

5.5 Efficiency and Decoding Dynamics

I summarize latency and beam behavior during decoding. Tables 5.18 show time per graph, pruning mix, and beam sensitivity. ACC checks are fast, and the CKG prior adds a sub-millisecond cost.

Table 5.18: 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

Most candidate expansions are eliminated by structural constraints, and control-flow reachability alone removes 54–63% (see Appendix D.2, Table D.1). The effective

branching factor remains low (about 2.1), which keeps the search efficient. Increasing the beam from $(4, 12, 3)$ to $(8, 24, 5)$ raises Validity by 4.2 points, IPA by 13.1 points, and LCS by 6.3 points, at an added 6.9 ms per graph (Appendix D.2, Table D.2). Latency is dominated by the encoder, and ACC overhead is minimal, so the default beam offers a good balance between performance and memory use.

5.6 Robustness and Generalization

Robustness is assessed on three ways: cross-project generalization, forward-in-time evaluation, and invariance to benign refactorings, using official project-disjoint `ReposVul` splits and averaging over five seeds. Across unseen projects, graph-level metrics shift minimally, with chain metrics degrading less than classification ones. Validity and interprocedural use (IPA) remain high, tied to static structure, while role coverage and order agreement track classification trends within overlapping confidence intervals. Calibration fixed on validation holds on test, with stable Expected Calibration Error and preserved PR/ROC rankings. Sink neutralization sharply lowers probabilities; guard strengthening causes small, directional changes. Chain validity stays high, indicating structural gate stability. Refactoring tests with renaming, inert code, and intra-block reorderings (all at 0.3 probability) show little effect except strong sensitivity to aggressive reordering. Overall, the chain-centric detector generalizes well, remains accurate over time, and tolerates harmless refactorings; see Appendix D for detailed deltas and intervals.

5.7 Interprocedural Root-to-Sink Chain Decoding

In this case study, I illustrate how the model behaves in practice by describing one decoded chain from the two-file program `main.c` and `lib/shell.c`, using the same trained model and ACC-constrained decoding with the default beam settings as in the main experiments. The full textual trace and exact source files are provided in Appendix G.11.

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`.

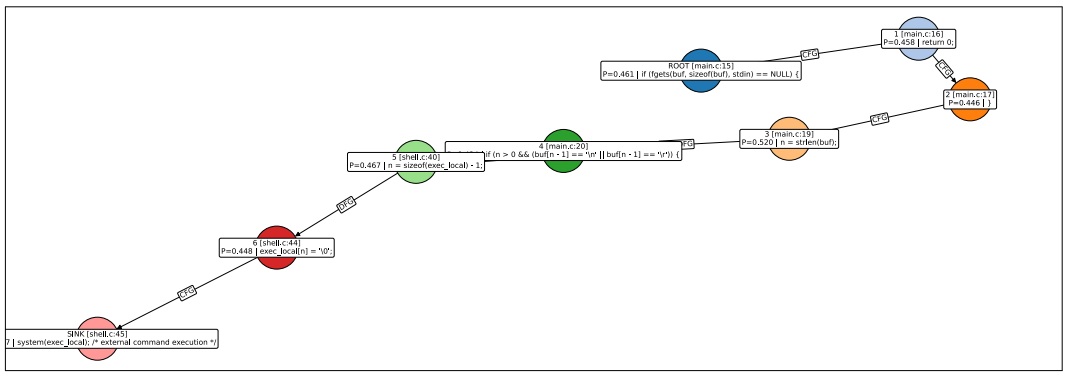

Figure 5.1: Multi file executable chain from root to sink.

5.8 Summary

The results in this chapter demonstrate competitive detection performance with strong AUPRC and F1 scores. Reconstructed chains show high feasibility and consistent interprocedural use concerning semantic roles and sequence order. 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 6

Conclusion and Future Work

6.1 Conclusion

This thesis began from the observation that real-world vulnerabilities rarely stem from a single line of code. Instead, they emerge from execution paths that move across multiple functions, files, and modules. Existing tools, both classical and data-driven, are effective at flagging suspicious regions but usually stop at local predictions. They seldom explain how untrusted data travels from its origin to an exploitable sink or how different parts of a program combine to create a vulnerability. The central goal of this work was therefore to move from local, correlation-based classification to the reconstruction of *executable interprocedural root-to-sink vulnerability chains* that a developer can follow and verify.

To achieve this, the thesis introduced an interprocedural representation that extends Code Property Graphs with explicit cross-function semantics, including calls, argument→parameter links, return→caller edges, and conservative alias relationships. On top of this graph, the approach combines structure-aware embeddings with relation-aware graph reasoning and a chain-focused decoding strategy. Adaptive Causal Contextualization (ACC) refines importance signals along control-flow and data-flow paths, and constrained beam search, guided by a compact Causal Knowledge Graph prior, assembles chains that are reachable, interprocedurally legal, and consistent with the underlying program semantics. Explanations are thus expressed as executable paths rather than isolated highlights.

The thesis also emphasized that high predictive performance is not sufficient on its own. Alongside standard detection metrics, it introduced causal and chain-centric evaluation criteria that test whether predictions change as expected under targeted edits and whether the model's attention is concentrated on the recovered chains. Experiments on real repositories show that this interprocedural, chain-centric view can improve robustness to code changes and produce explanations that are easier to interpret and more closely aligned with the steps required for remediation. In summary, the work demonstrates that vulnerability detectors can

be designed not only to say *where* a problem is likely to be, but to explain *how* and *why* it can be exploited through a concrete, executable chain.

6.2 Future Work

Although the proposed approach advances causality-aware vulnerability detection, there are several directions in which it can be extended and refined. One natural step is to move from a single-chain explanation per vulnerability to richer multi-chain outputs. In practice, a vulnerability may have several distinct exploit paths, or variants of the same path under different call sequences. Supporting multiple, diverse chains with well-controlled redundancy would give analysts a more complete picture of risk while still keeping explanations manageable.

Another direction concerns the priors and semantics that guide decoding. The current Causal Knowledge Graph is extracted once and used as a fixed prior at inference time. Making this component learnable and adaptive, for example by fitting it jointly with the main model or fine-tuning it per project, could improve alignment with local coding styles and domain-specific idioms. At the same time, enriching the interprocedural representation with more precise alias information, better handling of dynamic dispatch, and framework-aware summaries would make reconstructed chains more faithful to actual behaviour.

A third line of work is to incorporate lightweight dynamic evidence into this approach. Static analysis alone can admit paths that are technically possible but unlikely or infeasible in practice. Coverage data, symbolic execution traces, or feedback from fuzzing could be used to prioritize or filter chains, assign likelihood scores to segments, or confirm that a reported path is executable under realistic conditions. This would help bridge the gap between static reasoning and observed runtime behaviour.

The approach is currently developed and evaluated in a specific language setting(C/C++). Extending it to other languages, paradigms, and vulnerability types, including concurrency bugs, logic flaws, or vulnerabilities in multi-language stacks, would test the generality of the chain-centric perspective. In parallel, integrating the pipeline into developer workflows and running user studies would provide direct evidence of its practical value, for instance in terms of triage time, fix quality, and developer trust in the explanations.

Finally, there is a broader methodological opportunity around evaluation. The causal criteria introduced here are a first step toward standardized measures of chain quality and causal faithfulness. Future work could refine these metrics, compare them across different models, and relate them to human-centred outcomes. Shared benchmarks with chain annotations, clear splits, and agreed evaluation protocols would help the community to compare approaches more meaningfully and accelerate progress toward vulnerability analysis tools that are both effective and genuinely informative.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

[12] M. Freitag and Y. Al-Onaizan, *Beam search strategies for neural machine translation*, arXiv preprint arXiv:1702.01806, (2017).

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

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

[15] 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.

[16] 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.

[17] 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).

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

[19] 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.

[20] 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).

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

[22] 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.

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

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] S. Mandal, A. A. Sekh, and S. K. Naskar, *Solving arithmetic word problems: A deep learning based approach*, Journal of Intelligent Fuzzy Systems, 39 (2020), pp. 2521–2531.

[36] 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.

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

[38] A. F. T. Martins, G. Correia, V. Niculae, G. Poesia, O. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, *Structured Decoding for Constrained Generation*, in International Conference on Learning Representations (ICLR) 2022, 2022.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] 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.

[45] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Pearson, 4th ed., 2020.

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

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

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

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

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

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

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

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

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

Appendix A

Algorithms and Pseudocode

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

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. Initialize the beam as

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

with $T(\pi), C(\pi), G(\pi)$ derived from v_0.

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 $u \to v$;
    $\Delta \leftarrow \alpha \log \sigma(z_v) + (1 - \alpha)\, c_{u\to v}^{(r)}$;
    $S_{\text{ACC}}(\pi') \leftarrow S_{\text{ACC}}(\pi) + \Delta$;
      −role_penalty(π') + san_bonus(π');
        −len_penalty(π') − rep_penalty(π');
      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 successor 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 $(\Delta p, \Delta S_{\text{ACC}})$;

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

Appendix B

Model Architecture: Formal Equations and Training Objective

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.

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

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

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

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

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

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	$\mathsf{cfg\_ok}, \mathsf{dfg\_ok}, \mathsf{ipa\_ok}, \mathsf{alias\_ok}$ (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.

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

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

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

Selection.

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

Notation & Symbols for Chain Extraction

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

B.7 Training Objective: Equation

Composite loss.

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

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

Evaluation Metrics: Formal Definitions and Diagnostics

C.1 Calibration and Thresholding

C.1.1 Operating Points

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

C.1.2 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

C.3.1 Validity of Predicted Chains

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

C.3.2 Structural Fidelity

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

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

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

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

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

C.3.3 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

C.4.1 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 $s_i \in \{+1, -1\}$:

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

C.4.2 Causal Feature Attribution Measure (CFAM)

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

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

Attributions are normalized per graph.

C.5 Decoding Diagnostics

To report definitions for completeness.

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

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

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

Average chain length, interprocedural hop ratio, motif coverage@K, and the ACC rejection mix (CFG/DFG/IPA/alias) are summarized as descriptive statistics.

Appendix D

Extended Results and Hyperparameters

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

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

Software and hardware manifest

Python version, key libraries, and a lockfile with package hashes are included with the artifact for exact replay. Device model, VRAM, CPU cores, and RAM are recorded in a system info JSON.

D.2 Beam Search and ACC Diagnostics

The table 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.

Table D.1: Beam and ACC diagnostics under K=8, B=24, H=5 (per graph).

Quantity	Median	P90	Struct-only	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%

Beam Sensitivity (K, B, H)

The table reports validation metrics and per-graph latency for the GCBERT+Struct model across beam sizes.

Table D.2: 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

D.2.1 Decoder Diagnostics and Remedies

The table below lists rare decoding issues, their signals, and practical fixes.

Table D.3: 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

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 to patch histories and code before/after fixes. Each entry includes CVE/CWE identifiers, CVSS fields, patch commit metadata, and file contents (pre/post), aligned with the needs of interprocedural, chain-centric analysis.

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

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

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

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

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

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

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

Appendix F

Role Lexicon and Pattern Rules

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

F.1 API Families and Examples

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

F.2 Structural Patterns

1. Guards that dominate a sink and constrain tainted values (e.g., range checks on indices or lengths that post-dominate sources and dominate the sink).
2. Def–use chains that cross call boundaries via actual, formal bindings (ARG→PARAM; RET→CALLER/RET→LHS).
3. Alias-induced flows through pointers, references, and containers where points-to sets intersect along writes/reads.

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

Appendix G

Reproducibility Checklist and Artifact Details

This appendix lists the shipped artifacts and gives a concrete, step-by-step guide to rebuild graphs, train models, export chains, and regenerate every table and figure. It consolidates data provenance, graph-building configuration, embedding caches, training settings, environment controls for determinism, and exact commands for both Linux and Windows PowerShell.

G.1 Data provenance

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

G.2 Graph build configuration

Parsers and normalization: The graph builder records parser versions and normalization options (identifier anonymization, literal bucketing). Enabled relation families: AST/CFG/DFG; `CALL/RET/ARG2PARAM/RET2CALL/RET2LHS`; alias summaries. Slice parameters (diff windows, function/file/repository scopes) are stored with the same config.

G.3 Embeddings cache

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

G.4 Training configuration

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

G.5 Environment and determinism

Software and hardware: Experiments use PyTorch 2.4.1 with CUDA 12.1 on a single NVIDIA GeForce RTX 4070 Laptop GPU. Random seeds are fixed; deterministic/cuDNN flags avoid nondeterministic kernels where possible. Mixed precision uses dynamic loss scaling with critical reductions accumulated in FP32. An environment lockfile and package hashes are included with the release.

G.6 Artifact Overview

- Per-graph predictions and reconstructed chains (JSONL), with beam traces.
- Calibration parameters (temperature) and ECE bin assignments.
- Counterfactual intervention logs (pre/post probabilities, edit type, expected direction).
- Configuration files for each experiment (including seeds and hyperparameters).
- Environment manifests and package hashes.
- CKG prior database and mined motifs, with coverage summaries.
- Decoding diagnostics (per-hop scores and ACC rejection reasons).
- Scripts to regenerate graphs and recompute all metrics and figures.

G.7 Project layout (as shipped)

```
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 One-shot pipeline (shortcut)

This condensed sequence runs the full pipeline.

Linux

```
python -m pip install -r requirements.txt && \
python src/preprocess/step1_prepare_diversevul.py && \
python src/preprocess/step2_generate_pdg.py && \
python src/preprocess/step2c_pipeline_validate_and_slice.py && \
python src/preprocess/embed_lines_gcbert.py && \
python src/preprocess/slices_to_pyg_gcbert.py && \
python src/build_dataset_jsonl.py && \
python src/step4_train_gnn_attn.py && \
python src/step5_export_chains.py && \
python src/step6_eval_report.py
```

Windows PowerShell

```
python -m pip install -r .\requirements.txt
python .\src\preprocess\step1_prepare_diversevul.py
python .\src\preprocess\step2_generate_pdg.py
python .\src\preprocess\step2c_pipeline_validate_and_slice.py
```

```
python .\src\preprocess\embed_lines_gcbert.py
python .\src\preprocess\slices_to_pyg_gcbert.py
python .\src\build_dataset_jsonl.py
python .\src\step4_train_gnn_attn.py
python .\src\step5_export_chains.py
python .\src\step6_eval_report.py
```

G.10 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.11 Source for Figure 5.1 and Chain Trace

The files below reproduce Figure 5.1. 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.

G.12 How tables and figures are built

`step6_eval_report.py` aggregates per-graph outputs to CSV and LaTeX, mapping columns 1:1 to metrics in Chapter 4 and Appendix C. The report also emits PR curves, reliability diagrams, and distributions for chain length and interprocedural ratio under `Report/`.