Master's Thesis

# A Study of Synthetic-Data-Enhanced Analysis for Smart Contracts: Function-Level Detection and Explanation

NGUYEN Minh Ngoc

Supervisor NGUYEN Minh Le

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Master of Science (Information Science))

September, 2025

**Abstract**

Smart contracts are self-executing programs that run on blockchain platforms, most notably Ethereum. They automate transactions and enforce agreements without intermediaries, forming the foundation of decentralized finance (DeFi), non-fungible tokens (NFTs), and decentralized applications (dApps). Despite their growing importance, smart contracts remain prone to security vulnerabilities. Exploited bugs can lead to irreversible financial losses, service disruptions, and systemic failures. Although machine learning-based tools have emerged to aid vulnerability detection, two critical challenges remain: (1) limited fault localization at the function level, and (2) a lack of interpretable, human-readable explanations that enable developers to understand and fix issues effectively.

This thesis addresses both challenges by proposing a unified framework that combines graph-based neural network modeling with explainable language model techniques. Specifically, the contributions consist of: (1) a function-level vulnerability detection system using Sub-Graph Neural Networks (Sub-GNNs), and (2) an explanation generation mechanism based on synthetic data and Chain-of-Thought (CoT) prompting using large language models (LLMs). These two components aim to improve both the technical granularity and practical usability of smart contract security analysis.

The first part of the thesis introduces a novel function-level detection method that decomposes smart contracts into subgraphs centered around individual functions. While prior approaches using Graph Neural Networks (GNNs) operate at the contract level, they fail to pinpoint specific sources of vulnerabilities, limiting their value for debugging and remediation. To overcome this, we construct function-level subgraphs that incorporate control-flow and data-flow dependencies, preserving the semantic and structural context of each function. We then apply a Sub-GNN model to perform vulnerability classification at this finer granularity. Empirical evaluation on a curated synthetic dataset demonstrates that the proposed method achieves high precision in localizing faulty functions. Although it trades off a small margin of global classification accuracy compared to full-graph models, the localized predictions are significantly more actionable for developers. A benchmark comparison quantifies this trade-off and validates the effectiveness of subgraph-based analysis in practical settings.

To facilitate this line of work, we develop a synthetic dataset of smart contracts with function-level vulnerability labels. The dataset includes diverse vulnerability types such as reentrancy, integer overflows, access control flaws,

and unhandled exceptions. Each function is annotated with corresponding vulnerability types and contains metadata for constructing control and data flow graphs. This dataset fills a gap in the current landscape, which largely lacks fine-grained, labeled corpora for training and evaluating function-level detectors.

The second component of the thesis tackles the issue of explanation. While detecting a vulnerability is important, understanding why it occurs and how to resolve it is crucial for real-world usability. Most existing detection tools output low-level indicators such as line numbers or vulnerability labels without offering semantic explanations. To address this gap, we propose an explanation generation system that produces structured, human-readable justifications for detected vulnerabilities. We construct another synthetic dataset where each entry consists of a vulnerable function, its formal label, and a professionally formatted explanation describing the issue, its cause, and suggested remediation steps. These explanations are derived from real-world audit patterns and follow a consistent template.

Together, these two components form a comprehensive framework for smart contract vulnerability analysis. The Sub-GNN-based detector provides precise localization of faulty functions, while the CoT-guided explanation generator delivers semantic insight into the causes and consequences of the vulnerabilities. This dual capability bridges the gap between vulnerability detection and developer comprehension.

The thesis concludes with a discussion of future directions. On the detection side, extending the Sub-GNN architecture to support inter-function and inter-contract reasoning could enable the modeling of call chains and complex compositional vulnerabilities. On the explanation side, integrating user feedback to iteratively refine generated explanations could support interactive auditing tools. Furthermore, we propose exploring multimodal models that combine graph-based embeddings with textual features to enhance both detection and explanation tasks.

***Keywords***: Smart Contracts, Vulnerability Detection, Graph Neural Networks, Subgraph Analysis, Function-Level Classification, Chain-of-Thought Prompting, Large Language Models, Security Analysis, Solidity, Synthetic Dataset, Blockchain Security.

***Acknowledgment***

First and foremost, I would like to express my sincere gratitude to my academic supervisor, Prof. Nguyen Le Minh, for his invaluable guidance, continuous encouragement, and unwavering support throughout my time at JAIST. His deep insights, constructive feedback, and patience have been instrumental in shaping my research and thesis work. I am equally thankful to my second supervisor, Prof. Kiyoaki Shirai, and my minor supervisor, Prof. Naoya Inoue, for their helpful advice and support during the entire research process.

I am also profoundly grateful to all the faculty members, colleagues, and fellow researchers at Nguyen's Lab. Their collaborative spirit, technical discussions, and willingness to help have significantly enriched my research experience. The open exchange of ideas and the supportive environment in the lab have greatly enhanced my understanding and contributed to both my academic and personal growth. I extend my heartfelt thanks to all the professors at JAIST for providing such a stimulating and nurturing academic environment that has encouraged critical thinking and innovation.

Lastly, I owe my deepest gratitude to my family for their unwavering love and support. To my parents and sister, your belief in me, your sacrifices, and your constant encouragement have been the foundation of my motivation and perseverance. I could not have reached this milestone without your unconditional support and inspiration.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The rapid proliferation of blockchain technology has ushered in a new era of decentralized applications, where smart contracts-self-executing programs deployed on platforms like Ethereum-form the backbone of trustless computation [16]. These contracts enable automated financial operations, digital asset exchanges, governance mechanisms, and various decentralized finance (DeFi) protocols without the need for intermediaries. Smart contracts, primarily written in domain-specific languages such as Solidity, are immutable once deployed [22], meaning any flaws embedded in the code become permanent and directly exploitable. As a result, the security of smart contracts is a critical concern, with several high-profile exploits leading to substantial financial and reputational damage within the blockchain ecosystem.

Despite significant advancements in static analysis and machine learning–based vulnerability detection tools, two fundamental challenges continue to hinder robust and practical smart contract security analysis [13, 12, 19]. First, existing tools largely operate at the contract level, identifying whether a vulnerability exists somewhere in the contract without specifying the exact source of the problem. This lack of fine-grained fault localization limits the utility of detection tools for developers and auditors who must still manually investigate the contract to isolate the bug. Second, many tools lack explanatory power; they provide binary labels or low-level alerts without articulating the underlying reasoning or impact of the detected issue. Consequently, developers often receive insufficient guidance to understand or remediate the vulnerabilities found.

This thesis tackles both of these open problems by introducing a unified framework that leverages advances in graph-based deep learning and explainable artificial intelligence (XAI). The central goal is to improve not only the precision of smart contract vulnerability detection but also its interpretabil-

ity, thus aligning technical rigor with practical usability. The research is structured around two core contributions:

1. **Function-Level Vulnerability Detection via Sub-Graph Neural Networks (Sub-GNNs):** A novel method for identifying vulnerabilities at the granularity of individual functions rather than entire contracts. This is achieved by constructing semantic subgraphs using control and data flow analyses, allowing the detection model to focus on localized program behavior.

2. **Explanation Generation via Synthetic Data and Chain-of-Thought Prompting:** A systematic approach to produce high-quality, human-readable explanations for vulnerabilities, utilizing synthetic datasets enriched with detailed annotations and a structured prompting mechanism that enables large language models (LLMs) to generate interpretable reasoning steps.

The first component addresses the granularity gap in current vulnerability detection methods. While prior work in graph neural networks (GNNs) has shown promise in modeling smart contract semantics [25], these models typically analyze whole-contract graphs, making them ill-suited for pinpointing faults to specific functions. In contrast, this research decomposes each contract into per-function subgraphs that retain structural and semantic information. These subgraphs serve as the input to a Sub-GNN model trained to classify the vulnerability status of each function. Experimental results demonstrate that this approach enables precise fault localization, supporting more actionable insights for developers. Although it incurs a modest drop in global F1 score compared to contract-level methods, the ability to directly isolate vulnerable functions presents a significant usability improvement.

To facilitate this research, a synthetic dataset is constructed, comprising function-level Solidity examples with injected vulnerabilities across several common categories, including reentrancy, integer overflows, access control violations, and unchecked exceptions. Each instance is labeled and paired with its corresponding control/data-flow subgraph. This dataset, released alongside the thesis, serves both as a training resource and an evaluation benchmark, addressing a current gap in fine-grained vulnerability datasets for smart contracts.

The second component of this thesis centers on the interpretability challenge. Even when a vulnerability is correctly flagged, developers often struggle to understand its root cause or the rationale behind the detection. Current tools rarely explain why a given segment of code is vulnerable or how

it should be fixed [12, 13]. To fill this explanatory void, the thesis introduces a complementary explanation generation system built atop synthetic training data. Each data point pairs a vulnerable function with a structured, high-quality explanation that outlines the cause, implications, and suggested remediation of the vulnerability.

This explanation generation process is guided by CoT prompting, a technique that improves reasoning transparency in LLMs by encouraging step-by-step outputs. By decomposing the explanation into intermediate reasoning stages-such as identifying the vulnerable code path, characterizing the vulnerability type, and assessing its impact-CoT prompting enhances the clarity and actionability of the generated content. Evaluations using GPT-family models show that CoT-enhanced explanations significantly outperform baseline outputs in terms of coherence, readability, and usefulness to developers.

Together, these two contributions form a complete pipeline for vulnerability detection and understanding. The Sub-GNN framework identifies where vulnerabilities exist with fine granularity, while the explanation generation module provides insights into why they occur and how to address them. This dual emphasis on precision and interpretability aligns with the increasing demand for practical, developer-friendly security tools in the blockchain domain.

This research makes the following key contributions:

- A novel Sub-GNN–based model for function-level smart contract vulnerability detection.
- A publicly available synthetic dataset of annotated, vulnerable Solidity functions, along with their control/data-flow subgraphs.
- A high-quality explanation dataset tailored for vulnerability reasoning, constructed to support both supervised and generative learning tasks.
- An explanation generation framework using chain-of-thought prompting, empirically shown to improve explanation quality across multiple metrics.

By advancing both the granularity and interpretability of smart contract vulnerability analysis, this work aims to bridge the gap between automated detection and human-centered understanding, ultimately contributing to more secure and trustworthy decentralized applications.

# Chapter 2

# Related Work

## 2.1 Smart Contract Vulnerability Detection

### 2.1.1 Static Analysis Tools

Static analysis has been a foundational approach for detecting vulnerabilities in smart contracts, particularly those written in Solidity. These tools typically analyze source code or compiled bytecode without executing the contract, aiming to uncover potential vulnerabilities such as reentrancy, arithmetic overflows, or access control violations before deployment. Over the years, several prominent tools have been developed, each employing distinct analysis techniques and offering various levels of precision, coverage, and usability.

**Mythril** [4] is among the earliest and most widely used symbolic execution engines for Ethereum smart contracts. It translates bytecode into an intermediate representation and explores feasible execution paths using constraint solvers. Mythril can detect a range of common vulnerabilities, such as reentrancy and integer arithmetic bugs, by simulating different environmental conditions and control errors. While powerful in theory, symbolic execution suffers from path explosion and scalability limitations, particularly for contracts with complex state transitions or multiple functions

**Oyente** [7], one of the first vulnerability detectors for Ethereum, introduced a modular pipeline combining symbolic execution with control flow graph (CFG) construction to detect specific vulnerability patterns. Oyente was instrumental in demonstrating the feasibility of static analysis for smart contract security, but its relatively limited set of vulnerability checks and high false positive rates restricted its adoption in production environments.

**Securify** [8] approached the problem differently by formalizing security

properties as compliance and violation patterns within an intermediate representation of the contract. It uses Datalog queries to reason about contract behavior, offering a balance between precision and scalability. However, Securify's analysis is constrained by the expressiveness of its predefined property templates, which may fail to generalize to novel or compound vulnerability cases.

**Slither** [5] stands out for its developer-oriented design and extensibility. Built on a custom static analysis framework, it translates Solidity source code into a rich Abstract Syntax Tree (AST) and intermediate representations (e.g., SSA form). Slither offers a suite of prebuilt detectors, visualization tools, and taint tracking mechanisms. Its speed and flexibility make it suitable for integration into CI pipelines and auditing workflows. Nevertheless, its rule-based architecture can lead to both false positives and false negatives, especially in contracts that use unconventional programming idioms.

Table 2.1: Comparison of Static Analysis Tools for Solidity Smart Contracts

| Tool | Analysis Type | Granularity |
|---|---|---|
| **Mythril** | Symbolic Execution | Contract-level |
| **Oyente** | Symbolic Execution + CFG | Contract-level |
| **Securify** | Formal Verification (Datalog) | Contract-level |
| **Slither** | AST + Dataflow Analysis | Function-level |

While these tools have proven effective at detecting certain classes of vulnerabilities, they all suffer from a common set of limitations. First, they generally operate at the contract level, identifying whether a contract exhibits a vulnerability but failing to localize the exact source, such as a specific function or code segment. Second, their reliance on handcrafted rules or path exploration heuristics makes it difficult to adapt to previously unseen vulnerability types or code patterns. Third, most static tools offer limited explanatory capability-alerts are typically reported as warnings with minimal contextual information, leaving developers to infer the root cause and remediation steps manually.

These shortcomings motivate the development of machine learning–based techniques, particularly graph-based models, which can learn structural patterns of vulnerable code and generalize beyond predefined rules. However, static analysis tools remain a valuable baseline and often serve as the ground truth or annotation source for training and evaluating learning-based systems. In this thesis, we build upon these foundations by introducing a function-level detection model that improves the granularity and interpretability of smart contract vulnerability analysis.

### 2.1.2 Machine Learning Approaches

In recent years, machine learning (ML) has emerged as a promising alternative to traditional static analysis techniques for smart contract vulnerability detection [13, 12, 22]. Unlike rule-based or symbolic execution tools, ML-based models can learn patterns of vulnerability from labeled data, enabling generalization to previously unseen code structures and reducing reliance on handcrafted heuristics. These approaches typically fall into two broad categories: feature-based models and representation learning models.

Feature-based methods represent one of the earliest applications of ML to smart contract security. These approaches rely on manually engineered features extracted from source code, bytecode, or abstract syntax trees (ASTs). For instance, Zhu et al. [24] proposed using opcode sequences by replaying real-world transactions from the Ethereum Mainnet in a fully synchronized node, while the author leveraged a plugin called "SODA" to label opcode sequences with vulnerability classes and train a deep classification model using LSTM neural networks. Other work has leveraged features such as the number of external calls, the presence of certain control-flow structures, or data-flow reachability metrics [12, 13]. While these methods have demonstrated moderate success, their effectiveness is inherently constrained by the quality and completeness of the feature engineering process. Furthermore, these models struggle to capture complex semantic interactions between different parts of the code, especially in multi-function or compositional contracts.

To overcome these limitations, more recent work has focused on representation learning, particularly using neural networks to automatically learn latent code representations. One class of such models includes sequence-based deep learning approaches, such as Recurrent Neural Networks (RNNs) and Transformers, applied to tokenized contract code or opcode sequences. For example, ContractWard (Wang et al.,) [20] extracts bigram features from simplified operation codes. They then applied five machine learning algorithms combined with two sampling techniques, notably using XGBoost with SMOTETomek, to train efficient and accurate detection models. While sequence-based models are more flexible than traditional classifiers, they often fail to preserve structural information inherent in program logic, such as control and data dependencies.

A more structurally aware direction is the use of graph-based models, where smart contracts are represented as graphs-such as ASTs, CFGs, Program Dependency Graphs (PDGs), or custom interprocedural representations. Meanwhile, GNNs have shown strong performance in this setting. For example, Zhuang et al. [25] employ a Degree-free Graph Convolutional Neural Network (DR-GCN) and a Temporal Message Propagation Network

(TMP) to learn from the normalized graphs for vulnerability detection.

However, these models predominantly operate at the contract level, providing a binary vulnerability label for the entire contract. This coarse granularity limits their utility for real-world developers, who require precise localization of bugs for remediation. Moreover, most GNN-based models are trained on contract-level datasets, such as SmartBugs or curated subsets of Etherscan, which lack fine-grained annotations at the function or statement level.

In addition to detection, a few recent studies have explored multitask learning, combining vulnerability classification with auxiliary tasks such as type inference or code summarization, aiming to improve generalizability and interpretability. Others have experimented with code embedding techniques (e.g., Code2Vec [1], CodeBERT [9], or GraphCodeBERT [11]) to extract semantic representations of functions, which are then used in downstream classification tasks. While such methods introduce valuable inductive biases, they often treat code as flat sequences or bags of tokens, ignoring relational and hierarchical structures critical for vulnerability reasoning.

Despite these advancements, several challenges remain. First, there is a scarcity of fine-grained, well-annotated datasets that support function-level learning and evaluation. Second, many ML models act as black boxes, offering little to no explanation for their predictions. Lastly, the integration of learned detectors with developer workflows is still limited due to a lack of actionable feedback (e.g., why a function is vulnerable, what exact lines are problematic, or how to fix them).

Table 2.2: Summary of ML-Based Vulnerability Detection Approaches

| **Work** | **Input** | **Level** |
|---|---|---|
| Zhu et al. [24] | Opcode sequences | Contract |
| Wang et al. (ContractWard) [20] | Bigram opcodes | Contract |
| Zhuang et al. [25] | CFG/PDG graphs | Contract |
| Alon et al. (Code2Vec) [1] | AST paths | Function |
| Feng et al. (CodeBERT) [9] | Code tokens | Function |
| Guo et al. (GraphCodeBERT) [11] | Code+data flow | Function |

This thesis aims to address these limitations by proposing a Sub-Graph Neural Network (Sub-GNN) approach for function-level vulnerability detection. By decomposing contracts into semantically meaningful function-level subgraphs and training a GNN to classify vulnerabilities at this finer granularity, we enhance the utility of detection outputs for developers. Furthermore, by pairing detection with a dedicated explanation-generation module,

we extend beyond classification to deliver interpretable and actionable insights, helping bridge the gap between ML-based detection and practical smart contract auditing.

### 2.1.3 Subgraph-Based Vulnerability Detection

Subgraph-based techniques for software vulnerability detection represent an emerging direction that seeks to overcome the limitations of coarse-grained, contract-level analysis by introducing *finer-grained reasoning at the function or code-block level*. While the majority of existing graph-based models operate on entire program graphs such as full Control Flow Graphs (CFGs), ASTs, or Code Property Graphs (CPGs), they tend to provide only binary contract-level predictions, making them insufficient for pinpointing specific buggy regions.

Only a few studies to date have explored *subgraph-level vulnerability reasoning* in the context of smart contracts or software systems more broadly. These early-stage efforts typically treat functions or code regions as isolated subgraphs and attempt to classify their vulnerability status. For instance, some models extract *function-level subgraphs* based on syntactic or control/-data dependencies and feed them into lightweight graph encoders such as GCN or GAT. However, such approaches are still rare, and function-level datasets with vulnerability annotations are limited. *Line-level detection*, while explored in traditional bug localization tasks, remains underdeveloped in the security domain due to challenges in data labeling and structural consistency.

Subgraph-based reasoning has been more extensively explored in related domains, such as *malware detection*, *program repair*, and *bug localization*. In these areas, researchers have used subgraph matching to identify malicious patterns, refactorable segments, or fault-prone code regions. For example, subgraphs representing suspicious control flow motifs or resource misuse patterns have been used as templates for detecting malware variants. Similarly, in software fault localization, line-level or block-level subgraph extraction has enabled high-precision localization of logic faults. These techniques demonstrate that *subgraph abstraction can enhance interpretability, reduce input complexity, and support modular learning* principles that are increasingly relevant to vulnerability detection as models scale to larger codebases.

## 2.2 Dataset for Smart Contract Security

The availability of high-quality datasets is fundamental for developing and evaluating machine learning models for smart contract vulnerability detection. As highlighted in the survey by Feng et al. [3], datasets in this domain vary significantly in terms of source, labeling quality, vulnerability type coverage, and granularity.

Early datasets such as Etherscan-based collections consist of verified smart contracts scraped from the Ethereum blockchain. While these datasets are large-scale and diverse, they often lack explicit vulnerability annotations, requiring downstream tools like Mythril, Oyente, or Slither to label them heuristically. This indirect labeling approach may introduce noise and limits the utility of the dataset for supervised learning.

To address this, curated benchmark datasets such as SmartBugs Wild [6] and SolidiFI-Benchmark [10] have been introduced. SmartBugs Wild contains real-world contracts labeled using static analyzers and includes a wide range of vulnerability types such as reentrancy, timestamp dependence, and unchecked return values. However, the annotations are primarily contract-level, limiting their use for fine-grained vulnerability localization tasks. SolidiFI-Benchmark improves labeling fidelity by including expert-reviewed vulnerabilities across multiple tools, but still lacks consistent function-level granularity.

Another direction involves the construction of synthetic datasets, where vulnerabilities are systematically injected into otherwise benign contracts. This allows for precise control over the location and type of vulnerability, enabling reliable function-level or even statement-level supervision. Synthetic datasets have been successfully used to train and evaluate subgraph-based models, particularly for fine-grained fault localization and explanation generation. However, as noted in [3], synthetic datasets may introduce distributional bias if the injected patterns deviate significantly from real-world vulnerability characteristics.

Despite these advancements, several challenges remain. First, there is a lack of large-scale, open-source datasets with function-level annotations and paired explanations, which are crucial for training interpretable models. Second, existing benchmarks often focus on a narrow subset of vulnerabilities, limiting model generalizability. Finally, there is a need for standardization in evaluation protocols and labeling formats to ensure fair comparison across models.

To support this thesis, we introduce a synthetic, function-level dataset covering a diverse set of vulnerability types, including reentrancy, arithmetic overflows, access control, and exception mismanagement. Each function is

annotated with vulnerability labels, subgraph metadata, and a corresponding human-readable explanation. This dataset fills an important gap in the existing literature and enables systematic evaluation of both detection and explanation components in our proposed Sub-GNN framework.

## 2.3 Code Vulnerability Explanation

Explaining why a given code snippet is vulnerable is critical for enabling developers and security auditors to not only detect but also remediate security issues effectively. Recent research has explored a range of approaches for producing human-readable explanations of code vulnerabilities, ranging from rule-based systems to prompting large language models (LLMs). We categorize these efforts into two broad classes: traditional explanation approaches and language model-powered explanation.

### 2.3.1 Traditional Explanation Approaches

Prior to the advent of large language models, vulnerability explanation primarily relied on static rules, symbolic reasoning, or manually engineered heuristics. Traditional tools such as Slither, Oyente, and Mythril provide vulnerability labels along with metadata like affected line numbers, code patterns (e.g., unsafe external calls, unchecked return values), or execution traces. These outputs offer partial insight into the nature of the vulnerability but often fall short in terms of semantic clarity and contextual understanding.

Rule-based explanation frameworks typically hardcode vulnerability templates (e.g., "Unchecked low-level call on line X") or link to documentation of common weaknesses enumerations (CWEs). While effective for known bug patterns, these methods suffer from poor generalization and are limited in their ability to capture nuanced or novel security issues. Furthermore, the generated explanations tend to be terse, technical, and often incomprehensible to non-experts.

Several works in traditional software engineering have proposed using static analysis in tandem with fault localization metrics (e.g., suspiciousness scores or data-flow slicing) to guide explanation generation. However, these approaches are rarely targeted at smart contracts, and their outputs are generally diagnostic rather than explanatory in nature.

### 2.3.2 Language Model-Powered Explanation

Recent advances in LLMs have introduced new opportunities for generating rich, human-readable vulnerability explanations. Unlike static tools, LLMs can synthesize explanations by reasoning over both syntax and semantics, and they can incorporate natural language cues directly into their outputs.

Several recent studies [15, 17] have shown that models such as GPT-4, CodeGemma, and CodeLlama can not only detect vulnerabilities with high accuracy but also provide natural language rationales when guided by prompt engineering or CoT prompting. These methods typically involve framing the input code and task instructions in a way that elicits a structured explanation, covering aspects like the affected code region, vulnerability type, cause, and potential remediation.

In particular, Sultana et al. [17] report that CoT-style prompting improves the clarity and utility of explanations across LLMs. They experiment with one-shot and few-shot prompting strategies on real-world datasets (e.g., DiverseVul [2]) and observe that models like CodeGemma outperform earlier baselines in generating coherent justifications, achieving a recall of up to 87% for vulnerability detection and offering contextualized rationales in plain English. Similarly, Hou et al. [15] emphasize that LLMs fine-tuned for code tasks can be further adapted for explanation tasks by training them to generate vulnerability descriptions conditioned on annotated examples.

Despite these promising developments, several challenges remain. First, there is a lack of standardized datasets that pair code snippets with high-quality, function-level explanations. Second, evaluating explanation quality remains subjective and underexplored; common metrics such as BLEU or ROUGE fail to capture the functional adequacy or clarity of an explanation. Lastly, current LLMs still struggle with multi-function reasoning and compositional bugs, motivating further research on incorporating structural priors (e.g., ASTs or control/data flow graphs) into explanation pipelines.

This thesis builds upon this line of work by introducing a synthetic dataset that aligns function-level smart contract code with corresponding vulnerability explanations and by applying CoT-enhanced prompting strategies to generate interpretable and actionable outputs. Our results show that explanation-aware LLM prompting significantly improves both detection and developer usability.

# Chapter 3

# Methodology

## 3.1 Problem Statement

Let $\mathcal{C}$ denote a smart contract composed of a set of functions $\{f_1, f_2, \ldots, f_n\}$. Each function $f_i$ can be represented as a program graph $G_i = (V_i, E_i)$, where nodes $V_i$ correspond to syntax or semantic entities (e.g., statements, variables), and edges $E_i$ represent structural relationships such as control flow, data flow, or call dependencies.

We define two core tasks:

**1. Function-Level Vulnerability Detection.** Given a function-level graph $G_i$, the goal is to learn a function-level classifier

$$\mathcal{F}_\theta : G_i \to y_i \in \{0, 1\}$$

where $y_i = 1$ indicates that function $f_i$ is vulnerable, and 0 otherwise. The model $\mathcal{F}_\theta$ is parameterized (e.g., by a Subgraph Neural Network) and trained on a dataset $\mathcal{D} = \{(G_i, y_i)\}_{i=1}^{N}$, where $N$ is the number of labeled functions.

**2. Vulnerability Explanation Generation.** For functions where $y_i = 1$, the goal is to generate a natural language explanation $e_i = \mathcal{G}_\phi(G_i)$, describing:

- Bug Name: the vulnerability type,
- Affected Area: vulnerability localization,
- Description: description of the vulnerability,
- Impact Assessment: the severity of the vulnerability,

- Mitigations: potential fixes or mitigations.

Here, $\mathcal{G}_\phi$ is a sequence-to-sequence model (e.g., an LLM with Chain-of-Thought prompting) that maps a structural input graph (or its code representation) to a textual explanation. The goal is to design a model that accurately predicts vulnerable functions and generates clear, actionable explanations-bridging structural learning (via Sub-GNN) and natural language reasoning (via LLM-based CoT prompting).

## 3.2 Synthetic Dataset for Function-Level Analysis

Despite growing interest in applying machine learning to smart contract vulnerability detection, existing datasets are primarily limited to contract-level annotations, which significantly constrains the training of models capable of fine-grained analysis. To support function-level vulnerability detection-central to our problem formulation (see Section 3.1)-we construct a comprehensive synthetic dataset explicitly annotated at the function granularity. The dataset includes diverse vulnerability types (e.g., `Reentrancy`, `Access Control`, `Unchecked Low-Level Calls`), and each vulnerability is programmatically injected into semantically valid smart contracts with precise labeling of injection locations.

Our dataset construction process follows a structured pipeline comprising three major stages: (1) collection and filtering of clean contracts, (2) assembly of representative vulnerability patterns, and (3) automated injection of vulnerabilities into selected locations in the clean contracts. A high-level overview of the procedure is illustrated in Figure 3.1, and a formalized version of the injection process is shown in Algorithm 1.

### 3.2.1 Clean Contract Collection and Validation

We begin by acquiring a corpus of real-world, open-source smart contracts from Etherscan[1], a widely used Ethereum block explorer and repository for verified Solidity contracts. Etherscan provides access to source code, bytecode, ABI definitions, and metadata for publicly deployed smart contracts, making it a valuable resource for dataset curation.

To collect clean source code, we implement a custom web crawler and API client that systematically queries verified contracts via Etherscan's developer

[1] `https://etherscan.io/`

API[2]. The process includes the following steps:

1. **API Key Registration:** First, we register an account to obtain an API key for authenticated requests to the Etherscan API.

2. **Address Collection:** We collect contract addresses either by parsing newly deployed contracts block-by-block or by downloading public datasets like SmartBugs, which already include known contract addresses.

3. **Contract Source Retrieval:** For each address, we invoke the 'getsourcecode' endpoint:

   ```
   https://api.etherscan.io/api?module=contract
       &action=getsourcecode
       &address=<CONTRACT_ADDRESS>
       &apikey=<API_KEY>
   ```

   The response includes JSON-formatted fields such as 'SourceCode', 'ABI', 'ContractName', 'CompilerVersion', and 'OptimizationUsed'.

4. **Filtering for Solidity Contracts:** We retain only those contracts whose 'SourceCode' is non-empty and 'CompilerVersion' begins with `v0.4`, `v0.5`, `v0.6`, or `v0.8`, ensuring compatibility with modern static analysis tools and injection scripts.

5. **Source Code Preprocessing:** Retrieved contracts are stored in structured directories with filenames matching the contract address. Each file is normalized by removing Etherscan wrapping artifacts, resolving imports if possible, and verifying that the code compiles with `solc`.

Using this method, we collect and preprocess over 2,000 verified contracts. For this study, we focus on 2,188 contracts curated from the SmartBugs dataset [6], which provides on-chain verified Solidity contracts with metadata, ensuring reproducibility. To ensure the reliability of downstream vulnerability injection and label assignment, we rigorously filter out contracts that may contain latent or pre-existing vulnerabilities. We adopt a consensus-based filtering approach by applying three widely-used static analysis tools:

[2] `https://docs.etherscan.io/`

**1. Slither** [5] Slither is a static analysis framework for Solidity that performs vulnerability detection, optimization advice, and code metrics extraction. It uses an intermediate representation of Solidity called SlithIR.

- **Input:** A single Solidity source file (with resolved imports).
- **Command:** `slither <contract.sol> --detect-all`
- **Output:** Vulnerabilities are printed to stdout and can be redirected to JSON using the `--json` flag.
- **Format:** Includes issue type, location (line, column), and source mapping.

If any high-severity issues (e.g., 'reentrancy', 'arbitrary-from', 'tx.origin') are detected, the contract is excluded.

**2. Mythril** [4] Mythril is a symbolic execution engine for EVM bytecode that detects security issues such as integer overflows, callstack depth issues, and unguarded call instructions.

- **Input:** Solidity source file or EVM bytecode.
- **Command:** `myth analyze <contract.sol> -o json`
- **Output:** JSON report listing detected issues.
- **Format:** Each finding includes a title, severity, description, and source location.

Contracts producing critical issues (e.g., 'External Call to User-Supplied Address', 'SWC-107') are discarded.

**3. Oyente** [7] Oyente is one of the earliest tools for symbolic execution of smart contracts. Though less maintained, it still detects reentrancy and transaction-ordering bugs.

- **Input:** Compiled bytecode using `solc --bin`.
- **Command:** `python oyente.py -s <contract.sol>`
- **Output:** Console output with flags like 'Transaction Order Dependency', 'Reentrancy', or 'Time Dependency'.
- **Format:** Console printouts indicating specific vulnerability types and function locations.

Contracts flagged for transaction ordering or timestamp dependencies are excluded.

Subsequently, we follow strict filtering criteria and output formatting. To form a high-confidence clean set $C = \{c_1, c_2, \ldots, c_n\}$, we only retain contracts that:

- Successfully compile using `solc`;
- Are not flagged by any of the three tools above;
- Have valid source code with sufficient function-level granularity (i.e., `>2` function definitions).

Each clean contract $c_i$ is stored with metadata:

- `address`: Ethereum address of the deployed contract.
- `name`: Contract name (from AST).
- `source_path`: Path to the cleaned Solidity source file.
- `compiler_version`: Extracted from Etherscan metadata.
- `tool_report`: JSON object with scan results from Slither, Mythril, and Oyente.

These contracts form the basis of our injection dataset pipeline. The filtering process is deterministic and reproducible, and all scripts are made available with logging at each stage for future auditing and replication.

### 3.2.2 Vulnerability Snippet Curation

The next stage involves preparing a diverse and representative set of 287 vulnerable code snippets $S = \{s_1, s_2, \ldots, s_m\}$, each corresponding to a known vulnerability class $\mathcal{L}(s_i) \in \mathcal{S}$, where $\mathcal{S}$ comprises:

- `ACCESS_CONTROL`
- `ARITHMETIC`
- `DENIAL_OF_SERVICE`
- `FRONT_RUNNING`
- `REENTRANCY`

- TIME_MANIPULATION
- UNCHECKED_LOW_LEVEL_CALLS
- BAD_RANDOMNESS

These snippets are curated from the SolidiFI repository [10] and augmented with semantic annotations and structural constraints. Each snippet is suitable for direct injection, either as a source-code template or AST node, and is designed to activate realistic, semantically meaningful vulnerabilities.

**Access Control (ACCESS_CONTROL):** Access control vulnerabilities result from missing or improperly implemented authorization checks. The injected snippets often remove `require(msg.sender == owner)` or introduce unguarded state-modifying logic (e.g., transferring funds or modifying critical variables). When injected, they allow unauthorized users to invoke privileged functions, affecting the global integrity of the contract and breaking assumed trust boundaries. This compromises functions like `mint`, `updateConfig`, or `selfDestruct`, leading to privilege escalation and unauthorized control.

**Arithmetic Bugs (ARITHMETIC):** This class covers integer overflows and underflows, especially in older Solidity versions or in unprotected calculations without the `SafeMath` library. Typical snippets include unchecked operations like `balance -= amount` or `count++`. Post-injection, these bugs silently introduce numeric inconsistencies, which may cascade into allocation errors, faulty loop conditions, or token mismanagement. They impact any logic that depends on numeric values, making validation and assertions unreliable across functions.

**Denial of Service (DENIAL_OF_SERVICE):** DoS vulnerabilities occur when a contract's logic can be blocked or indefinitely stalled. Common patterns include looping over unbounded storage arrays or failing to handle external call failures. Injected snippets emulate this by, for example, inserting unbounded loops over dynamic data structures or storage-intensive logic inside frequently called functions. Once injected, these vulnerabilities affect availability, making functions non-executable under gas limits and stalling business logic, especially during mass operations like token distribution.

**Front Running (FRONT_RUNNING):** Front-running vulnerabilities stem from the predictability and public visibility of transaction ordering. Injected snippets demonstrate exploitable state updates based on externally visible variables such as `msg.value`, `tx.origin`, or prior storage reads. When injected,

they allow attackers to anticipate and preemptively act upon sensitive state changes (e.g., by sandwiching transactions). This introduces fairness issues, particularly in time-sensitive auctions, token sales, or liquidity pool updates, and corrupts the expected execution ordering semantics.

**Reentrancy (`REENTRANCY`):** Reentrancy is introduced by inserting unsafe low-level calls (e.g., `recipient.call.value(amount)("")`) before updating the contract's internal state. Upon injection, such snippets allow external contracts to recursively invoke the same function before the original call completes, leading to duplicated withdrawals or corrupted balances. The injected vulnerability affects all state-manipulating logic sharing the same storage and fundamentally violates the atomicity assumptions of contract execution.

**Time Manipulation (`TIME_MANIPULATION`):** This vulnerability arises when critical contract logic depends on timestamps or block numbers that can be influenced by miners. Injected code patterns include `require(now < unlockTime)` or `if (block.timestamp % 10 == 0)`. Once injected, these enable adversaries to manipulate time-based access control, locking mechanisms, or randomness seeds. As a result, the injected contract may permit early withdrawals, extend auctions unfairly, or bypass timed constraints-thus undermining time-sensitive logic across the contract.

**Unchecked Low-Level Calls (`UNCHECKED_LOW_LEVEL_CALLS`):** Snippets in this category add low-level calls without checking the return value (e.g., `addr.call.value(x)("")` without a success check). Post-injection, the contract continues execution under the false assumption that the call succeeded, causing silent failures. These bugs often impact fund transfers, callback mechanisms, or event-triggered chains, breaking error-handling semantics and causing inconsistent state transitions or silent value loss.

**Bad Randomness (`BAD_RANDOMNESS`):** Randomness vulnerabilities emerge from using manipulable sources like `block.timestamp`, `blockhash`, or `block.difficulty` as entropy. Injected snippets demonstrate insecure generation patterns such as

```
uint r = uint(keccak256(abi.encodePacked(
    now, blockhash(block.number - 1))))
```

Such randomness can be predicted or influenced by miners or front-runners. Once injected, the randomness becomes attackable, affecting lotteries, game logic, and any probabilistic decision-making in the contract.

Each injected vulnerability is crafted to preserve the compilability of the host contract and to localize the defect to function-level regions. Despite localized injection, these vulnerabilities often exhibit cascading side effects-altering interprocedural control flow, invalidating invariants, and weakening systemic safety guarantees. Our approach ensures that each modified contract $c_i'$ contains both semantic and structural annotations, enabling function-level supervision and model evaluation for granular vulnerability detection.

### 3.2.3 Injection and Annotation via SolidiFI

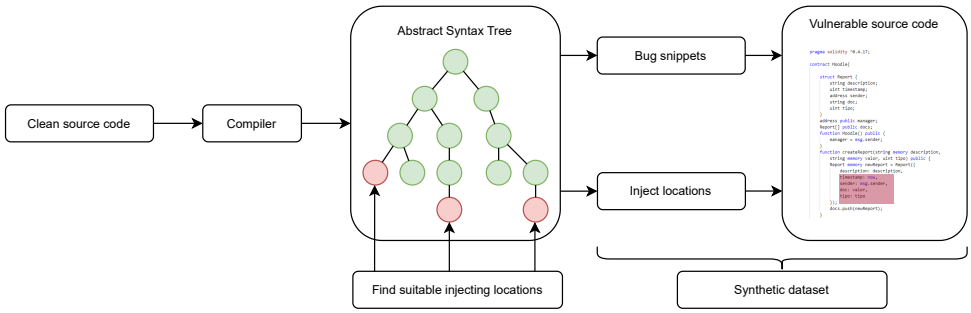


Figure 3.1: Bug Injection with SolidiFI

The process of injecting vulnerabilities and annotating smart contracts operates on a filtered corpus of verified, clean contracts $C = \{c_1, c_2, \ldots, c_n\}$ and a curated set of vulnerability snippets $S = \{s_1, s_2, \ldots, s_m\}$. We follow a systematic pipeline to transform these clean contracts into a labeled dataset $C' = \{c_1', c_2', \ldots, c_k'\}$, where each $c_i'$ contains exactly one injected vulnerability along with metadata describing the injection. The process is reflected in Figure 3.1 and Algorithm 1.

The first step is to ensure that the original contracts are free of pre-existing vulnerabilities. For this, we apply a consensus-based filtering strategy using static analysis tools $\mathcal{T}$, namely Slither, Mythril, and Oyente. Each contract $c \in C$ is scanned by all tools. If no known vulnerability is reported by any tool, the contract is considered clean and added to a filtered set $C_{\text{clean}} \subseteq C$. This ensures that injected bugs can be confidently attributed to our injection process, without interference from latent flaws in the source.

Once we obtain the filtered set $C_{\text{clean}}$, we initialize the output dataset $C' \leftarrow \emptyset$. For each contract $c \in C_{\text{clean}}$, we begin by flattening the contract if it spans multiple source files. This step uses `slither-flat` under the `OneFile` strategy. We determine the root file by analyzing the contract's import hierarchy and either matching the declared `ContractName` or selecting

**Algorithm 1** Vulnerability Injection Process for Creating a Labeled Smart Contract Dataset

**Require:**

- $C = \{c_1, c_2, \ldots, c_n\}$: Set of clean smart contracts.
- $S = \{s_1, s_2, \ldots, s_m\}$: Set of vulnerable code snippets from SolidiFI.
- Static analysis tools $\mathcal{T}$: e.g., Slither, Mythril, Oyente.

**Ensure:**

- $C' = \{c'_1, c'_2, \ldots, c'_k\}$: Set of synthetic vulnerable contracts with corresponding vulnerability labels.

**Clean Contract Filtering**$(C, \mathcal{T})$:
  **for each** contract $c \in C$ **do**
    **if** not `is_vulnerable`$(\mathcal{T}, c)$ **then**
      Add $c$ to filtered set $C_{clean}$.
    **end if**
  **end for**

**Initialize** $C' \leftarrow \emptyset$ {Set of injected, labeled contracts}
**Vulnerability Injection**$(C, S)$:
  **for each** contract $c \in C_{clean}$ **do**
    Parse $c$ with SolidiFI to obtain $AST$ and potential injection points.
    $c$.`potential_inject_locations` $\leftarrow$ `SolidiFI`$(c)$
    **for each** position $p \in c$.`potential_inject_positions` **do**
      **if** $p$.`is_low_level()` **then**
      **Sample** $s \in S$.
      **Insert** $s$ at $p$.`src` of smart contract $c$.
      **Compile and verify** the modified contract $c'$.
      **if** $c'$ is compilable **then**
        **Label** $c'$ with the vulnerability type of $s$ and the injection location(s).
        **Add** $c'$ to dataset $C'$.
      **else**
        **Discard** $c'$; try a new snippet $s$ or a new injection point $p$.
      **end if**
    **end if**
  **end for**
**return** $C'$ {Final dataset of labeled, vulnerable contracts}

the most-imported file in the tree. After flattening, the unified Solidity file is saved to a dedicated directory for processing.

Next, we compile the flattened contract using `solcx` to generate its Abstract Syntax Tree (AST). The compiler version is selected according to metadata obtained from the original contract, and the full AST output is stored as a JSON file. We then parse this AST to enumerate potential injection positions. These are typically statement-level nodes within functions or modifiers that satisfy certain properties (e.g., visibility is `public` or `external`, or the node is a control-flow block). Each injection point $p$ is tagged with its source location via the `src` field, formatted as `start:length:file_index`, which facilitates precise source-code rewriting.

For each injection point $p \in c.\texttt{potential\_inject\_positions}$, we randomly sample a vulnerability snippet $s \in S$. Each snippet is designed to represent one of the vulnerability classes described in Section 3.2.2 (e.g., reentrancy, bad randomness, access control). Before injection, we verify that $s$ is structurally compatible with the context at $p$, including checks for variable scoping, type safety, and syntactic validity. If compatible, we insert $s$ directly at the location specified by $p.\texttt{src}$ within the source string of $c$.

After insertion, we recompile the modified contract $c'$ using the same `solcx` configuration. If the compilation fails-due to syntax errors, missing declarations, or unresolved imports-the contract is discarded, and we retry with a different snippet or location. If compilation succeeds, we proceed to annotate $c'$ with relevant metadata. Each entry is labeled with the vulnerability type $\mathcal{L}(s)$, the name of the function or block where the injection occurred, the precise `src` coordinates, and a copy of the inserted code snippet. This information is stored in both human-readable CSV and machine-parseable JSONL formats.

This process is repeated iteratively over all contracts in $C_{\text{clean}}$. The output is a function-level labeled dataset of synthetic smart contracts with real-world codebases and realistic injection semantics. Each sample in $C'$ is guaranteed to be compilable, traceable to a known vulnerability, and useful for training and evaluating detection models at fine granularity.

## 3.3 Function-Level Vulnerability Detection Via Sub-Graph Neural Networks (Sub-GNNs)

### 3.3.1 Subgraph Neural Network Architecture

Our proposed model, denoted `FunctionLevelSubGCN`, is a subgraph-aware neural architecture designed to classify vulnerability presence at the function

level. It operates on annotated Abstract Syntax Graphs (ASGs) representing each contract and leverages both global graph structure and localized subgraph context to improve fine-grained detection.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the ASG of a smart contract, where each node $v \in \mathcal{V}$ is associated with a node type embedding $x_v \in \mathbb{R}^{d_{\text{in}}}$ and each edge $(u, v) \in \mathcal{E}$ encodes either structural, semantic, or data dependencies. For each function $f_i$ in the contract, we define a function-level subgraph $\mathcal{G}_{f_i} = (\mathcal{V}_{f_i}, \mathcal{E}_{f_i}) \subseteq \mathcal{G}$.

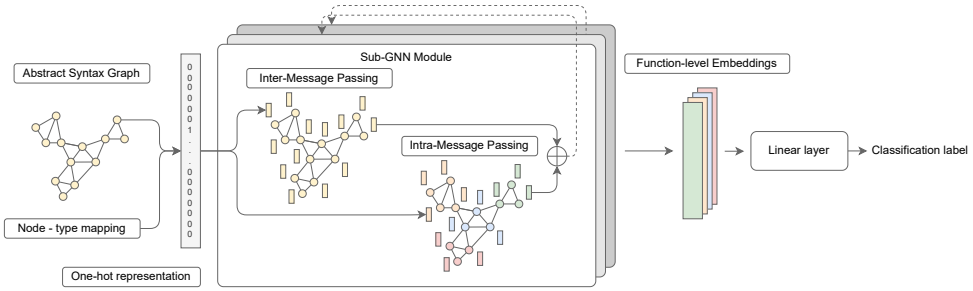


Figure 3.2: Sub-Graph Neural Network Architecture

The input to the model includes:

- A node feature matrix $X \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{in}}}$, where $d_{\text{in}} = 44$ denotes the one-hot encoded node type dimension.
- A global edge index $\mathcal{E}_{\text{global}}$ for full-ASG message passing.
- A subgraph edge index $\mathcal{E}_{\text{sub}}$ for function-specific message passing.
- A subgraph assignment vector $\mathbf{g} \in \{1, \ldots, G\}^{|\mathcal{V}|}$ mapping each node to its function subgraph.

**Dual-Channel Message Passing.** The model consists of $L = 3$ GNN layers. At each layer $l$, it performs parallel message-passing over:

1. The global ASG $\mathcal{G}$ using $\mathcal{E}_{\text{global}}$ to capture cross-function dependencies.
2. The local function subgraphs $\mathcal{G}_{f_i}$ using $\mathcal{E}_{\text{sub}}$ to capture intra-function control/data flow.

The main difference between inter- and intra-message passing layers is illustrated in figure 3.3

- In inter-message passing, edges can cross between functions (messages flow across functions).

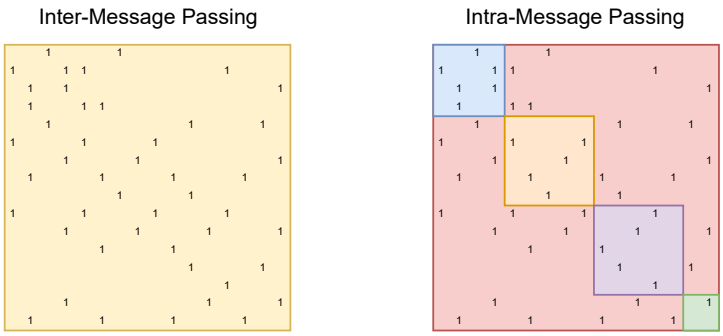


Figure 3.3: Inter- versus Intra-Message Passing

- In intra-message passing, only edges inside each function remain while the rest are masked. That means each function forms an isolated subgraph where GAT attention only operates locally.

Each message-passing operation uses a `GATv2Conv` layer. The global propagation result at layer $l$ is given by:

$$H_{\text{full}}^{(l)} = \texttt{GATv2Conv}_{\text{full}}^{(l)}\left(H^{(l-1)}, \mathcal{E}_{\text{global}}\right), \tag{3.1}$$

and the subgraph-local propagation is:

$$H_{\text{sub}}^{(l)} = \texttt{GATv2Conv}_{\text{sub}}^{(l)}\left(H^{(l-1)}, \mathcal{E}_{\text{sub}}\right), \tag{3.2}$$

where $H^{(0)} = X$ is the initial node embedding matrix. The results from both channels are concatenated and passed through a linear transformation and nonlinearity:

$$H^{(l)} = \text{ReLU}\left(W^{(l)} \cdot \left[H_{\text{full}}^{(l)} \parallel H_{\text{sub}}^{(l)}\right]\right), \tag{3.3}$$

where $W^{(l)} \in \mathbb{R}^{2d \times d}$ projects the concatenated embeddings back to the hidden dimension.

**Subgraph-Level Pooling.** After $L$ propagation layers, each node embedding $h_v^{(L)}$ is mapped to a corresponding function via the subgraph membership vector $\mathbf{g}$. A custom max pooling function $\texttt{maxpool}(\cdot)$ is applied per function:

$$z_i = \max_{v \in \mathcal{V}_{f_i}} h_v^{(L)}, \tag{3.4}$$

where $z_i \in \mathbb{R}^d$ is the pooled embedding representing function $f_i$. This aggregation captures the most salient semantic features of the function by preserving the maximum response across all nodes in the subgraph.

**Prediction Head.** The pooled embedding $z_i$ is passed through a final linear layer to produce class logits:

$$\hat{y}_i = W_{\text{out}} z_i + b_{\text{out}}, \tag{3.5}$$

where $W_{\text{out}} \in \mathbb{R}^{2\times d}$ and $b_{\text{out}} \in \mathbb{R}^2$. The output $\hat{y}_i$ is then interpreted as the logit vector over the binary classes: `vulnerable` and `non-vulnerable`. Optionally, a dropout-activated linear block with ReLU may be applied before this layer for regularization.

**Loss Function.** The model is supervised using cross-entropy loss over all function-level predictions:

$$\mathcal{L} = -\sum_{f_i \in \mathcal{C}} y_i \log \sigma(\hat{y}_i^{(1)}) + (1 - y_i) \log(1 - \sigma(\hat{y}_i^{(1)})), \tag{3.6}$$

where $y_i \in \{0, 1\}$ is the ground truth label, and $\sigma$ denotes the softmax-normalized class probability.

The architecture incorporates hierarchical message propagation by learning both full-graph (contract-wide) and subgraph-local (function-scoped) embeddings through dedicated GNN channels. By combining attention-based convolution, subgraph-aware pooling, and function-level supervision, the model can excel at pinpointing vulnerabilities in the correct semantic scope, especially for interaction-induced bugs that span across multiple functions. This design is also modular and extensible, allowing for the future integration of inter-contract dependencies or richer type-aware node features.

### 3.3.2 Semantic Augmentation via Pretrained Function Embeddings

While the Sub-GNN architecture effectively captures intra- and inter-function structural dependencies, certain vulnerability patterns may span beyond syntactic graph structure and depend on latent semantic cues present in function names, control logic, or developer conventions. To incorporate such high-level semantic information, we introduce an additional embedding stream derived from pretrained function-level embeddings. Specifically, we employ the `Salesforce/codet5p-110m-embedding` model [21], a lightweight encoder-only variant of CodeT5+, to generate fixed-length representation vectors for each function.

We adopt `codet5p-110m-embedding` due to its favorable trade-off between performance and efficiency. It comprises only 110M parameters, supports fast inference through an encoder-only architecture, and has been pretrained on a diverse corpus of programming languages, including Solidity.

This model is open-source, robust, and suitable for embedding code fragments without requiring downstream fine-tuning. Its architecture is designed for embedding-based retrieval and classification tasks, making it a compelling choice for static code analysis in resource-constrained settings.

**Embedding Extraction and Fusion.** During preprocessing, we tokenize each function's source code and extract its corresponding semantic embedding using the CodeT5+ model. Each resulting vector is of dimension 256 and is stored alongside the graph-structured data. After the message-passing phase of the Sub-GNN, we apply custom max pooling over node embeddings to generate one vector per function, yielding a set of function-level structural representations. These pooled vectors are concatenated with the corresponding CodeT5+ semantic embeddings to form hybrid representations:

$$z_i = \texttt{ReLU}(\mathbf{W}[x_i \,\|\, e_i]),$$

where $x_i$ is the graph-based subgraph representation of the $i$-th function, $e_i$ is the CodeT5+ embedding, $\|$ denotes concatenation, and $\mathbf{W}$ is a learnable linear transformation matrix mapping the joint feature to the hidden space.

The fused embeddings $z_i$ are passed through an additional feedforward layer and projected into the output space via a final linear classifier. This produces logits over vulnerability labels for each function, integrating both structural and semantic signals. This dual-stream approach enhances the model's ability to detect cross-function or semantically obfuscated vulnerabilities that may not manifest in purely graph-local structures.

Certain vulnerabilities-such as those involving improper use of access modifiers or inconsistent state updates-may involve functions that are not directly connected in the abstract syntax graph but share semantic relationships through naming, parameter patterns, or shared responsibilities. The incorporation of pretrained embeddings helps capture such latent relationships and augments the Sub-GNN with a richer contextual understanding. Furthermore, the low computational footprint of `codet5p-110m-embedding` makes it a practical solution for scaling to large datasets without sacrificing performance or interpretability.

## 3.4 Vulnerability Explanation Via Synthetic Data and Chain-of-Thought Prompting

### 3.4.1 Bug Explanation Synthesis

The next essential part of this work involves the generation of structured and interpretable vulnerability explanations. These explanations bridge the gap between automated detection and actionable remediation, facilitating not only understanding but also further supervised training of explanatory models. The primary objective of this stage is to generate consistent and technically accurate descriptions of vulnerabilities using a large language model, specifically GPT-4o.

To ensure consistency, reduce ambiguity, and enforce semantic alignment between the label and the content of the explanation, the system utilizes a structured prompt formulation strategy. Each data instance is represented as a pair

$$\mathcal{V} = (\mathcal{C}, y)$$

, where $\mathcal{C}$ denotes the Solidity code containing the vulnerability, and $y$ is the ground-truth label indicating the type of vulnerability, selected from a predefined taxonomy $\mathcal{Y}$ (e.g., `reentrancy`, `access_control`, `unchecked_low_level_call`, etc.). The input to GPT-4o is a triplet

$$\mathcal{P} = (\text{Instruction}, \mathcal{C}, y)$$

, where the instruction explicitly requests a vulnerability analysis in a fixed output format. To minimize the risk of misclassification during generation, the label $y$ is supplied directly to GPT-4o as part of the prompt. This helps constrain the model's generative space and focus its reasoning process on the relevant vulnerability type.

The generation is guided by a structured template, which defines a structured template for the explanation: *Bug Name*, *Affected Area*, *Description*, *Impact Assessment*, and *Mitigation*. Figure 3.4 presents the specific format used in the prompting process. The prompt instructs GPT-4o to assume the role of a smart contract security expert and produce an explanation following this structured schema. This format was designed to balance technical completeness with human readability, ensuring that each explanation captures the essential aspects of the vulnerability while remaining interpretable to both expert and non-expert audiences.

A core component of this methodology is the use of **CoT prompting**, which instructs GPT-4o to reason through the explanation in a step-by-step

**Structured Template**

You are a smart contract security expert. Analyze the following Solidity code snippet and explain the vulnerability.
The vulnerability type is [Correct Vulnerability Label]. Use the following format:

### Bug Name: [Correct Vulnerability Label]
### Affected Area:
### Description:
### Impact Assessment:
### Mitigation:

Figure 3.4: Structured Template for Bug Explanation Synthesis

manner. Rather than producing an answer in a single pass, the CoT framework encourages the model to decompose the task into smaller reasoning units. The model begins by parsing the provided code snippet, identifying relevant functional blocks such as fallback functions, state variable declarations, and external calls. It then isolates critical program locations likely to be associated with the vulnerability, denoted as

$$L_v \subset \mathcal{L}(\mathcal{C})$$

, where $\mathcal{L}(\mathcal{C})$ is the set of all logical statements within the contract. The model proceeds to trace how the vulnerable behavior arises by identifying violations of best practices, such as failing to update internal state before interacting with external addresses or lacking appropriate access controls.

By enforcing this decomposition, CoT prompting enhances the transparency of the reasoning process. For example, in the case of a reentrancy vulnerability, GPT-4o will first identify the external call occurring within the function, then note the order of operations, and finally reason about how recursive invocation can lead to repeated balance withdrawals before the state is updated. The model concludes by suggesting concrete mitigations, such as the adoption of the Checks-Effects-Interactions pattern or the inclusion of OpenZeppelin's `ReentrancyGuard`. A full example of the output generated under this schema is shown in Figure 3.5, where GPT-4o demonstrates a structured and technically sound breakdown of the vulnerability, its consequences, and remediation.

The explanation generation function can be formally expressed as

$$f_{\text{explain}}(\mathcal{C}, y) \rightarrow \mathcal{E}$$

**GPT-4o's Example Output with Structured Template**

### Bug Name
Reentrancy Vulnerability

### Affected Area
The 'withdraw' function in the contract.

### Description
The 'withdraw' function transfers Ether to an external address before updating the contract's state. This allows an attacker to recursively call the 'withdraw' function before the balance is deducted, enabling them to drain the contract's funds.

### Impact Assessment
This vulnerability can lead to significant financial losses, as attackers can exploit it to steal all Ether stored in the contract. It is a critical issue that compromises the integrity and security of the smart contract.

### Mitigation
To mitigate this vulnerability, apply the Checks-Effects-Interactions pattern. Ensure that the contract's state is updated before transferring Ether to external addresses. Additionally, consider using reentrancy guards or the 'ReentrancyGuard' utility provided by OpenZeppelin.

Figure 3.5: GPT-4o's Example Output with Structured Template

, where $\mathcal{E}$ is the structured explanation output comprising the five prescribed sections. This generation process is stateless and parallelizable, making it suitable for large-scale data generation pipelines. In practice, multiple vulnerability instances can be processed concurrently through batched API calls to GPT-4o, achieving high throughput and significantly reducing the time and labor traditionally associated with manual explanation writing.

Scalability is a key component of this methodology. It takes a lot of resources to manually analyze and document smart contract vulnerabilities, and auditors may not always do this consistently. On the other hand, thousands of thorough explanations with consistent structure and quality can be produced using the automated method outlined here. This is especially useful for training models that benefit from explanation-aware supervision or for research applications involving sizable smart contract corpora.

In addition to directing the model's focus, the explicit inclusion of the

ground-truth label $y$ in the prompt reduces the possibility of hallucinations, which are a frequent problem in generative language models when incorrect or unsupported information is introduced. The approach improves the generated content's semantic fidelity by tying the explanation to a recognized vulnerability type. This method, however, is predicated on the accuracy of the labels supplied during the data synthesis stage. The significance of accurate labeling during data generation is highlighted by the fact that any labeling noise added upstream will spread throughout the explanation pipeline.

A smart contract security analysis pipeline's downstream components can directly use the explanations that are produced. They will be used, for example, as training data to improve language models that learn to generate explanations for vulnerabilities from code. As an alternative, they can be incorporated into auditing platforms' user interfaces to help security experts evaluate possible threats. The output can also be programmatically incorporated into structured reports or documentation systems since it follows a predetermined schema.

### 3.4.2 Fine-tune Open-Source LLMs

To adapt a general-purpose code language model for the specific task of vulnerability explanation, we conducted supervised fine-tuning on the CodeLlama-13b-hf model. The objective was to align the model's autoregressive generation behavior with the structured output format required for smart contract bug explanation, leveraging a synthetic dataset composed of labeled Solidity examples.

The dataset used in this stage was generated by injecting 287 distinct vulnerability code snippets into a corpus of 2,188 clean smart contracts. Each vulnerability snippet corresponded to a known bug type drawn from a predefined taxonomy. The injection process resulted in 8,101 executable contracts after filtering out contracts that failed to compile or execute correctly. This final dataset was then partitioned into three disjoint subsets: $\mathcal{D}_{\text{train}}$ (80%), $\mathcal{D}_{\text{val}}$ (10%), and $\mathcal{D}_{\text{test}}$ (10%), ensuring that validation and testing were conducted on unseen samples. Each training instance is represented as a tuple $(x, y) \in \mathcal{D}_{\text{train}}$, where $x$ is a Solidity source code snippet and $y$ is the corresponding textual explanation. For each contract, the target output follows a five-field structured schema: *Bug Name*, *Affected Area*, *Description*, *Impact Assessment*, and *Mitigation*.

Prior to training, a preprocessing pipeline was applied to the Solidity inputs. This included stripping inline and block comments, normalizing indentation, and ensuring consistent syntax formatting. These steps were necessary to reduce variance in code style and simplify the mapping between

code and explanation during sequence modeling. The final tokenized sequences were constrained to a maximum length of 4096 tokens, with padding and truncation applied as needed using the CodeLlama tokenizer.

The fine-tuning process employed the **HuggingFace Transformers** framework and used the `CodeLlama-13b-hf` base model, chosen for its high parameter count and pretraining on a diverse corpus of source code, including Solidity. The training objective minimized the cross-entropy loss over the target tokens $y$, conditioned on the input prompt $x$. Let $\hat{y}_t$ denote the model's predicted probability distribution at time step $t$, and $y_t$ the ground-truth token. The loss function is defined as:

$$\mathcal{L} = -\sum_{t=1}^{T} \log \hat{y}_t[y_t]$$

where $T$ is the sequence length of the output.

The training was done with mixed precision and used **Low-Rank Adaptation (LoRA)** to ensure computational efficiency. This technique limits its weight updates to low-rank subspaces of the original model parameters. Without sacrificing representational power, this method enabled model refinement with much lower memory usage and training time. Only a portion of the transformer blocks' projection layers were altered by the LoRA setup, thereby freezing most of the parameters.

The **AdamW** optimizer was used for optimization, with cosine annealing for adaptive learning rate scheduling and a learning rate initialized at $\eta_0 = 3 \times 10^{-4}$. Depending on the training hardware's effective memory availability, training took place over five epochs with a dynamic batch size of eight samples. Early stopping was implemented based on the validation loss, which was tracked following each epoch, in order to avoid overfitting.

Following training, the optimized model was assessed on the held-out test set using the same human-aligned evaluation framework as GPT-4o explanations. Three criteria were specifically used to evaluate each generated explanation: **Clarity**, which refers to the output's grammatical coherence and semantic fluency; **Identification**, which indicates the correctness of the identified vulnerability and its location in the code; and **Impact**, which evaluates how well the explanation explains the possible repercussions of the bug and its implications on contract behavior. An overall interpretability score was generated by averaging the ratings of these dimensions on a 5-point Likert scale.

The refined CodeLlama-13b-hf model yielded explanations with a quality that was comparable to that of GPT-4o, but with significantly lower inference cost and latency, according to empirical results. This shows that employing

optimized open-source models for high-throughput, readable security analysis is feasible. Furthermore, its output's interpretability makes it a good fit for automated vulnerability detection pipeline integration, especially in situations where scalability and explainability are both crucial.

### 3.4.3 Evaluate The Explanations

We use a robust large language model, **GPT-4o**, as an automatic evaluator across a set of structured, semantically grounded metrics in order to evaluate the performance of explanation-generation models both quantitatively and qualitatively. The evaluation framework specifically concentrates on three fundamental dimensions: *Impact*, *Identification*, and *Clarity*. These standards were selected in order to fully capture each generated explanation's linguistic fluency, technical precision, and contextual relevance. By assigning evaluation to GPT-4o, we leverage its domain expertise and sophisticated reasoning skills to execute consistent, high-fidelity judging at scale.

A CoT format is used to prompt GPT-4o for each explanation in the test set. By dividing each evaluation task into logically sequential subquestions, this design promotes methodical analysis. The model is specifically instructed by the prompt to read the Solidity code snippet, look at the related vulnerability label, and methodically evaluate the explanation content. In addition to increasing evaluation consistency, this multi-step prompting structure makes the model externalize its reasoning process, which reduces erroneous or hallucinogenic judgments.

The explanation's linguistic quality is assessed by the first dimension, *Clarity*. From beginning to end, GPT-4o evaluates the text's logical structure, semantic coherence, and grammar. It also looks at how well the explanation conveys technical insights to a wide range of readers, from inexperienced developers to security experts, and whether it stays away from superfluous jargon. This guarantees that the content produced is accurate, pedagogically useful, and easily accessible.

The technical correctness of the explanation in respect to the related code and ground-truth label is the focus of the second dimension, *Identification*. GPT-4o checks to see if the explanation correctly identifies the type of vulnerability, the pertinent lines or functions in the codebase, and the prerequisites for exploiting the vulnerability. In order to demonstrate both internal consistency and external validity against the labeled source code, the explanation must demonstrate a type of semantic alignment. GPT-4o serves as an automated auditor during this stage, confirming that the model accurately depicts the attack vector and causal relationships in the contract.

The third dimension, *Impact*, gauges how thoroughly and precisely the

explanation takes into account the vulnerability's practical repercussions. GPT-4o assesses if the explanation includes a workable mitigation strategy and if it describes believable outcomes, such as monetary loss, denial-of-service, or data leakage. Crucially, this metric evaluates whether the recommended mitigations are workable, in line with industry norms, and suitable for the contract's structure in addition to simply listing the consequences.

For each explanation $e_i$ in the test set $\mathcal{D}_{\text{test}}$, the model assigns a numeric score in the range $[1, 5]$ to each of the three dimensions:

$$\text{Score}(e_i) = \left(s_i^{\text{clarity}}, s_i^{\text{ident}}, s_i^{\text{impact}}\right), \quad s_i^* \in \{1, 2, 3, 4, 5\}$$

The final evaluation metric is the arithmetic mean of these scores, representing the overall interpretability and utility of the explanation:

$$\text{Aggregate}(e_i) = \frac{s_i^{\text{clarity}} + s_i^{\text{ident}} + s_i^{\text{impact}}}{3}$$

This framework enables scalable and rigorous evaluation without the resource constraints of human annotation. Moreover, by relying on CoT reasoning, the system provides traceable justifications for each score, allowing researchers to audit and interpret the evaluation process itself. As a result, this methodology ensures that generated explanations are not only syntactically well-formed but also semantically rich and practically actionable-criteria that are essential for the deployment of explainable vulnerability detection in real-world smart contract auditing scenarios.

# Chapter 4

# Experiments and Analysis

## 4.1 Vulnerability Classification Task

In the present research, we test our model on two primary datasets: (1) a synthetic dataset created by inserting bugs into clean contracts, and (2) a real-world benchmark of 135 smart contracts with verified vulnerabilities. Function-level labels and ASTs for every contract are extracted from both datasets.

Vulnerable contracts from `SmartBugs` and `Solidifi` are combined in the real-world dataset. The marker `@vulnerable_at_lines` is used to annotate vulnerability locations directly in the source code for `SmartBugs`. This marker is then parsed to produce function-level bug labels. The accompanying CSV log files for `Solidifi` contain bug information that includes line numbers, bug types, and span lengths. Each contract's vulnerable code segments are identified using these annotations.

Clean contracts from the `SmartBugs Wild` repository are injected with predefined vulnerability templates to create the synthetic dataset. As explained in Section 3.2, each injection mimics common vulnerability classes like `reentrancy`, `access_control`, `arithmetic`, `time_manipulation`, `denial_of_service`, `unchecked_low_level_calls`, and `front_running`. This method guarantees that a wide variety of representative security issues are present in the dataset. The `getAST()` function is used to process all contracts, synthetic or real, in order to extract their AST. The contract's path, source code, list of bugs (if any), AST, bug type, and data source (e.g., `solidifi`, `smartbugs`, `clean`) are all included in each datapoint.

All processed data is serialized into a `.pkl` file (`ast_data.pkl` or `ast_data_trimmed.pkl`), which serves as input for model training and evaluation. Both synthetic and real-world contracts can benefit from fine-grained

vulnerability detection at the function level thanks to the consistent data representation.

### 4.1.1 Large Language Models vs. Sub-GNN on Function–Level Detection

To contextualize the effectiveness of our *function-level* Sub-GNN, we benchmarked it against two state-of-the-art large language models (LLMs) specialised for code understanding: **DeepSeek-Coder-V2-Lite-Instruct** and **Qwen 2.5-7B-Instruct**. Each LLM was queried in a strict zero-shot setting with the following instruction:

**Zero-shot prompt for function-level classification**

Given a piece of Solidity code, can you tell me which functions contain vulnerabilities?
Please provide the list of vulnerable functions as a Python list.
The Solidity code is as follows:
[REDACTED SOLIDITY CODE]

Figure 4.1: Zero-shot prompt for function-level classification

This prompt was appended with the full Solidity contract and submitted to the models without any fine-tuning or in-context examples.

To assess the effectiveness of our Sub-GNN architecture at identifying vulnerable functions in real-world smart contracts, we conducted a comparative evaluation against state-of-the-art large language models (LLMs) designed for code understanding and generation. For this experiment, we employed the SmartBugs Curated benchmark dataset, which consists of 234 manually labeled real-world smart contracts, with vulnerability annotations provided at the function level. This benchmark provides a realistic and challenging setting for evaluating fine-grained detection models.

The comparison results are shown in Table 4.1. Our semantic-augmented GNN variant, **SubGCN-FLE** (explained in Section 3.3.2), achieved the highest overall F1-score of **0.6720** with a strong precision of **0.9545**, indicating that the majority of functions predicted as vulnerable were indeed correct. However, its recall was relatively low (0.5185), suggesting that while it was highly conservative and accurate in its positive predictions, it missed a number of actual vulnerable functions. This behavior is consistent with the model's high-confidence thresholding, driven by the discriminative power of function-level embeddings.

Table 4.1: Function-level Detection Performance compared with Large Language Models.

| Model | Precision | Recall | F1 |
|---|---|---|---|
| SubGCN-FLE | **0.9545** | 0.5185 | **0.6720** |
| SubGCN-FLA | 0.5455 | 0.4074 | 0.4664 |
| DeepSeek-Coder-V2-Lite-Instruct | 0.1801 | 0.5506 | 0.2715 |
| Qwen2.5-7B-Instruct | 0.1812 | **0.7911** | 0.2948 |

The **SubGCN-FLA** variant, which aggregates local node features via dual-channel message passing without semantic embeddings (see Section 3.3.1), showed significantly lower performance, with an F1-score of 0.4664. This drop highlights the importance of incorporating semantic priors into graph-based representations, as purely structural models may lack the discriminative capacity to distinguish nuanced vulnerabilities.

In contrast, the LLM baselines underperformed on this task. **DeepSeek-Coder-V2-Lite-Instruct** achieved a precision of only 0.1801, with recall at 0.5506 and F1-score of 0.2715. Similarly, **Qwen2.5-7B-Instruct** reached higher recall (0.7911)—indicating it often marked vulnerable functions—but had very low precision (0.1812), resulting in many false positives and a final F1-score of 0.2948. These results reflect a common issue with instruction-following LLMs in code reasoning tasks: they tend to overgeneralize and generate plausible-sounding but imprecise outputs when not fine-tuned or guided by rigorous examples.

### 4.1.2 Function-Level Comparison with Existing GNN Baselines

State-of-the-art GNN baselines for smart-contract analysis generally operate at the *contract* granularity rather than the function granularity. To enable a fair comparison, we aggregate our function-level predictions: a contract is labelled *vulnerable* if *any* constituent function is predicted as vulnerable.[1] Table 4.2 summarises the F1, recall, and precision of our variants and of widely cited baselines on the same synthetic benchmark.

With **SubGCN-FLE (Function-Level Embedding)**, by leveraging the semantic augmentation detailed in Section 3.3.2, this variant achieves the highest F1 (0.9830) and near-perfect precision (0.9957). The seman-

[1]This aggregation mirrors industry practice where a single exploitable function compromises the entire contract.

Table 4.2: Models' performance on Synthetic Benchmark

| Model | Granularity | Sem. Aug. | F1 | Recall | Precision |
|---|---|---|---|---|---|
| SubGCN-FLE (ours) | Fine | Yes | **0.9830** | 0.9708 | **0.9957** |
| SubGCN-FLA (ours) | Fine | No | 0.9710 | 0.9624 | 0.9798 |
| TMP [25] | Coarse | No | 0.9510 | **0.9948** | 0.9109 |
| TransfomerGNN [23] | Coarse | No | 0.9372 | 0.9886 | 0.8909 |
| DRGCN [25] | Coarse | No | 0.9366 | 0.9793 | 0.8975 |
| GCN [14] | Coarse | No | 0.9121 | 0.9886 | 0.8466 |
| GAT [18] | Coarse | No | 0.9109 | 0.9845 | 0.8476 |
| SubGCN-CLA (ours) | Coarse | No | 0.9079 | 0.9845 | 0.8423 |

tic embeddings disentangle subtle usage patterns (e.g., re-entrancy gates vs. benign external calls), yielding very few false positives. **SubGCN-FLA (Function-Level Aggregation)** relying solely on dual-channel message passing (Section 3.3.1), FLA still surpasses every external baseline, confirming that explicit aggregation of per-function signals is already a powerful prior. **SubGCN-CLA (Contract-Level Aggregation)** also relying only on dual-channel message passing, but the last aggregation layer is max pool aggregation from all nodes in the graphs. This approach yield the lowest performance. Other **Baseline GNNs** like TMP and TransformerGNN obtain high recall but trail in precision, suggesting that their coarse-grained representations over-generalise vulnerable patterns. Classic GCN and GAT architectures exhibit the largest precision deficits, aligning with prior reports that purely based on contract-level aggregation is not sufficient for analyzing smart contracts.

Overall, the results underscore two conclusions: (i) semantic-aware embeddings elevate precision to industrially acceptable levels without sacrificing recall, and (ii) a function-level perspective, when intelligently aggregated, yields state-of-the-art contract-level detection while retaining the localisation benefits crucial for remediation workflows.

## 4.2 Vulnerability Explanation Task

### 4.2.1 Explanation Performance Benchmark

We performed a comparative benchmark analysis across seven models, both commercial and open-source, in order to thoroughly assess the caliber of vulnerability explanations produced by different large language models (LLMs). To guarantee a fair and thorough assessment, the chosen models reflect a va-

riety of architectural underpinnings, parameter scales, and training regimens. Notably, *open-source models* like *CodeLlama* and *Magicoder* were chosen to evaluate the performance of publicly available, code-specialized models under various levels of fine-tuning, while *commercial models* like *GPT-4o* and *Claude-3-Haiku* were included as state-of-the-art general-purpose LLMs with known competence in code understanding.

As fundamental baselines, *CodeLlama-7B* and *CodeLlama-13B* shed light on the ways in which explanation capability is impacted by model scale. We can also directly measure the impact of targeted training on explanation quality thanks to the inclusion of *CodeLlama-13b-hf-finetuned*, which has been fine-tuned on our domain-specific synthetic dataset. The open-source spectrum is further expanded by *Magicoder-S-DS-6.7B* and *Magicoder-S-CL-7B*, which are made for tasks involving code reasoning and instruction following. *GPT-4o* is the state-of-the-art in secure code analysis and natural language understanding, while *Claude-3-Haiku* is a lightweight substitute among commercial offerings.

Table 4.3: Performance comparison of different models in terms of clarity, identification, and impact. The **bold** values indicate the highest scores in each column.

| Model | Average | Clarity | Identification | Impact |
|---|---|---|---|---|
| CodeLlama-7b-hf-base | 2.7899 | 2.6087 | 2.9739 | 2.7870 |
| CodeLlama-13b-hf-base | 3.0014 | 2.6870 | 3.5304 | 2.7870 |
| CodeLlama-13b-hf-finetuned (ours) | 3.6638 | 3.2696 | **4.5696** | 3.1522 |
| Magicoder-S-DS-6.7B-base | 3.8333 | 3.6783 | 4.2174 | 3.6043 |
| Magicoder-S-CL-7B-base | 3.6971 | 3.5609 | 4.0261 | 3.5043 |
| claude-3-haiku-20240307 | 3.3710 | 3.6130 | 2.7174 | 3.7826 |
| GPT-4o | **4.0261** | **4.1783** | 3.6957 | **4.2043** |

With an average score of 4.03, **GPT-4o** performed the best overall, as indicated in Table 4.3. Its outstanding performance in **Clarity** (4.18) and **Impact** (4.20) is the main factor contributing to this high average. The high clarity score of GPT-4o indicates that its explanations are successful in making technical material understandable in addition to being grammatically correct and logically organized. According to its impact score, GPT-4o excels at explaining the wider ramifications of vulnerabilities, like monetary loss or user exploitation, and frequently offers doable mitigation techniques.

It's interesting to note that, despite GPT-4o's superior clarity and impact, the fine-tuned **CodeLlama-13b-hf-finetuned** outperformed it in the **Identification** category, earning an astounding 4.57. This finding supports

the theory that fine-tuning on a domain-specific explanation dataset significantly improves a model's capacity to identify and precisely characterize the location and technical nature of smart contract vulnerabilities. A targeted training process allows for more accurate vulnerability reasoning, even in the absence of the wide generalization capabilities of commercial LLMs, as evidenced by the fine-tuned model's strong identification performance.

The **Magicoder** family of open-source models performed well overall, particularly in **Clarity** and **Identification**. *Magicoder-S-DS-6.7B* outperformed even GPT-4o in the identification category, scoring 4.21 and 3.68 in clarity. This performance suggests that Magicoder's training on a variety of code problems and instruction-following tasks translates well to real-world vulnerability explanation tasks, particularly in situations where domain adaptation is not feasible. Notwithstanding their advantages, both Magicoder variations were still inferior to GPT-4o in the **Impact** category. This could be because they had a less thorough comprehension of the wider ramifications of security vulnerabilities, which may call for more general-purpose reasoning skills.

With an average score of just 2.79, **CodeLlama-7B** showed limited effectiveness on the lower end of the performance spectrum. The relatively lower identification and clarity scores imply that smaller-scale models find it difficult to balance the demands of both natural language articulation and technical reasoning. Though it still trailed behind refined and commercial alternatives, **CodeLlama-13B** outperformed its 7B counterpart, particularly in identification (3.53), suggesting some scaling benefit.

Finally, the performance profile of **Claude-3-Haiku** was not entirely consistent. It received the second-highest impact score (3.78) and strong clarity (3.61), demonstrating a strong capacity to convey wider ramifications. However, in line with its more general-purpose training goal, its low identification score (2.71) suggests a weakness in low-level technical reasoning.

In conclusion, the benchmark results show a definite trade-off between domain-specific fine-tuned models and general-purpose LLMs. **GPT-4o** is ideal for end-user-oriented explanation tasks because it raises the standard for impact and clarity. However, fine-tuned open-source models such as **CodeLlama-13b-hf-finetuned** provide better technical identification, indicating that hybrid approaches that combine prompt engineering and fine-tuning may provide the best results in tasks involving the explanation of smart contract vulnerabilities.

### 4.2.2 Evaluation of GPT-4o Score

We manually examined 50 randomly selected outputs in order to evaluate the accuracy of GPT-4o's automated assessments in the vulnerability explanation task. Human experts independently annotated these samples in three evaluative dimensions: **Impact**, **Identification**, and **Clarity**. Finding out how well GPT-4o's scoring matched human judgment in terms of both absolute score values and ordinal ranking was the aim of this assessment. In order to achieve this, we used two statistical correlation metrics: **Kendall's Tau**, which measures agreement in ranking order, and the **Pearson correlation coefficient**, which captures linear relationships between scores. Table 4.4 displays the findings.

Table 4.4: Correlation coefficients between manual and GPT-4o evaluations.

| | **Clarity** | **Identification** | **Impact** |
|---|---|---|---|
| Pearson | 0.100 | 0.860 | 0.321 |
| Kendall's Tau | 0.062 | 0.708 | 0.310 |

The **Identification** dimension showed the highest correlation, with GPT-4o achieving a Kendall's Tau of 0.708 and a Pearson coefficient of 0.860. Strong linear and ordinal agreement with human annotations is indicated by these values, indicating that GPT-4o is very good at detecting the existence, kind, and location of vulnerabilities in smart contract code. The structured nature of vulnerability identification, where factual correctness—like identifying the bug type (e.g., reentrancy, arithmetic overflow) and pointing to the pertinent code lines—can be objectively verified, is responsible for this reliability. Additionally, the application of standardized explanation formats and chain-of-thought prompting probably improved GPT-4o's accuracy and focus during assessment. The model's usefulness for scalable vulnerability triage in automated pipelines was demonstrated when it was given enough context and syntactic cues to consistently match expert reasoning in identifying security flaws.

Performance in the **Clarity** dimension, on the other hand, showed significantly less agreement. The evaluations of GPT-4o demonstrated little correlation with human assessments of linguistic quality and readability, with a Pearson score of just 0.100 and a Kendall's Tau of 0.062. Human evaluators commonly pointed out problems like excessive verbosity, inconsistent logical flow, and overuse of domain-specific jargon, even though the generated explanations were frequently technically correct. These flaws made the explanations less understandable and less instructive, particularly for users

with different skill levels. This disparity emphasizes the inherent difficulty of automating qualitative evaluations that involve audience sensitivity and subjective language features, which are less bound by formal structure.

With correlation values of 0.321 (Pearson) and 0.310 (Kendall's Tau), GPT-4o demonstrated a moderate level of alignment with human evaluators in the **Impact** dimension. The model frequently fell back on broad severity ratings that lacked nuanced risk contextualization, even though it showed a respectable ability to infer possible outcomes of vulnerabilities. For instance, rather than taking into account mitigating conditions like limited access control scope or non-critical contract logic, GPT-4o often assigned high-impact labels to bugs like reentrancy or unchecked low-level calls based on their theoretical exploitability. However, when assessing the practical implications of each vulnerability, human evaluators took into account real-world context, including asset exposure, user privileges, and contract deployment conditions. According to this gap, GPT-4o still lacks the domain-specific calibration required for context-aware risk prioritization, even though it is capable of reasoning over abstract threat models.

In conclusion, GPT-4o is a good fit for scalable code auditing and static analysis augmentation because it shows great dependability in objective, structurally based tasks like vulnerability identification. However, when dealing with subjective aspects that rely on understanding real-world context or subtle linguistic quality, its performance deteriorates. These results suggest that in order to guarantee accuracy and usability, hybrid evaluation workflows are currently required, in which GPT-4o performs preliminary assessments and human-in-the-loop refinement follows. Future research could concentrate on improving GPT-4o's performance in subjective evaluations by using context-aware risk modeling, feedback-tuned fine-tuning loops, and instruction tuning datasets made to make security explanations easy to understand.

## 4.3 Error Analysis: Limitations in Vulnerability Explanations

To better understand the strengths and limitations of large language models (LLMs) in vulnerability explanation, we performed a qualitative error analysis using an example smart contract function vulnerable to an *Unhandled Exception*. This analysis compares the outputs of two representative models: our fine-tuned **CodeLlama-13B** and the commercial **GPT-4o**. Figures 4.2 and 4.3 illustrate the explanations each model produced for the same buggy

**CodeLlama-13B-finetuned's explanation on an Unhandled Exception bug case**

### Bug name
Unhandled Exception

### Affected area
The affected area is the 'callee.call.value(1 ether)("")' line within the 'call_checked_lankford_transferFrom' function.

### Description
The vulnerability arises from the use of 'callee.call.value(1 ether)("")' which sends 1 ether to the 'callee' address without checking if the call was successful. In Solidity, the low-level '.call' method returns a boolean indicating whether the call was successful or not. However, in this code snippet, the return value of the call is not checked. This means that if the call fails (e.g., the 'callee' address reverts or runs out of gas), the subsequent lines of code will still execute. This can lead to unexpected behavior and potentially severe security issues.

### Impact assessment
The impact of this vulnerability is significant. If the call to the 'callee' fails, the function will still proceed to execute the 'transferFrom' and emit events. This can lead to inconsistent states, where tokens are transferred or approved despite the failure of the initial call. This can cause significant financial losses or disruptions in the contract's operations.

Figure 4.2: CodeLlama-13B-finetuned's explanation on Unhandled Exception bug

code segment.

The results show significant behavioral variations between the models. The main problem, which is the lack of a success check following the invocation of the low-level call, is accurately identified by the explanation produced by CodeLlama-13B-finetuned (Figure 4.2). This behavior demonstrates how well the model understands Solidity semantics, particularly how it handles call failures and the security implications that follow. Furthermore, the explanation shows a thorough and targeted comprehension of smart contract exception handling pitfalls. It does not, however, offer any specific mitigation, such as including a 'require(success)' check. This omission raises the possibility of a flaw in the model's fine-tuning data, which might have un-

**GPT-4o's explanation on an Unhandled Exception bug case**

### Vulnerability: Reentrancy Attack

### Affected Area:
The affected area is the call to ‘callee.call.value(1 ether)("")‘ before any state changes are made (e.g., updating the token balance).

### Vulnerability Description:
The function ‘call_checked_lankford_transferFrom‘ transfers 1 ether to an external contract (‘callee‘) and then attempts to transfer tokens by calling ‘_tokenImpl.transferFrom‘ function. The problem arises because the external call (‘callee.call.value(1 ether)("")‘) transfers control to the ‘callee‘ contract without any checks to ensure the ‘callee‘ contract is not malicious.
If the ‘callee‘ contract has a fallback function, that function gets executed upon receiving ether. If that fallback function calls back into the ‘call_checked_lankford_transferFrom‘ function or another function within the same contract that affects the state (e.g., modifying balances), it can lead to undesired reentrant calls. This can result in the contract being in an inconsistent state, and potentially exploited to drain ether or perform unauthorized transfers.

### Assessment of the Impact:
The impact of this vulnerability can be severe. If exploited, a malicious ‘callee‘ contract could utilize reentrancy to drain ether from this contract, conduct unauthorized transfers, or manipulate the internal state. This could lead to significant financial loss or the loss of control of the contract logic.

### Solution to Fix the Vulnerability (Mitigation):
To mitigate this vulnerability, state-changing operations should be performed before transferring control to external contracts. Specifically, move the critical operations (balancing updates and emitting events) before making the external call. Here's how to fix the vulnerable code: [REDACTED SOLIDITY CODE]

Figure 4.3: GPT-4o's explanation on Unhandled Exception bug

deremphasized remedial training or the explanation format's "mitigation" section.

On the other hand, GPT-4o incorrectly classifies the vulnerability as a reentrancy attack (Figure 4.3). This is a semantic error because, despite the external call's potential as a vector, the code doesn't include any state changes before the external call, which is one of the prerequisites for reentrancy to be exploitable. This misclassification suggests that GPT-4o relies

too much on surface-level patterns, where a reentrancy diagnosis is made just by the presence of an external call. However, by incorporating a clearly stated mitigation strategy, GPT-4o offers a more developer-focused and actionable explanation. It specifically suggests moving state updates prior to the external call and implementing the Checks-Effects-Interactions (CEI) pattern, which is a generally accepted best practice in Solidity development.

It's interesting to note that both models show failure modes associated with abstraction and context length. The security implications of the original bug may be exacerbated by CodeLlama's tendency to truncate or ignore surrounding interactions, such as token transfers ('transferFrom'). However, because it attempts to infer attack scenarios (like recursive fallback behavior) that aren't stated explicitly in the code, GPT-4o's more general contextual reasoning can occasionally result in hallucinated risks. The models' architectural biases are reflected in this divergence: GPT-4o excels at integrating high-level reasoning, albeit at the expense of occasional technical errors, while CodeLlama prioritizes syntactic precision over holistic context.

All things considered, this case study demonstrates the complementary advantages and drawbacks of both models. Although CodeLlama-13B-finetuned exhibits accurate vulnerability identification based on language-level comprehension, it is devoid of proactive repair techniques. Despite its propensity for overgeneralization, GPT-4o provides more thorough, developer-friendly answers along with mitigation recommendations. Future enhancements might include adding carefully chosen mitigation examples to CodeLlama's training data and tightening type and condition constraints to GPT-4o's diagnosis logic to prevent misclassification.

# Chapter 5

# Conclusion

This thesis combines language model-based explanation generation with graph-based learning to provide a comprehensive framework for enhancing the accuracy and interpretability of smart contract vulnerability analysis. It fixes two long-standing flaws in current tools: the lack of actionable, human-readable explanations for issues found and the coarse granularity of contract-level vulnerability detection. This work offers a dual improvement to the developer-facing usability of automated security analysis tools, encompassing both detection and interpretability.

Regarding detection, we suggest a new Sub-Graph Neural Network (Sub-GNN) model that can classify vulnerabilities at the function level. Compared to conventional contract-level GNNs, the model more accurately localizes vulnerabilities by breaking down smart contracts into function-level subgraphs built from control and data flow dependencies. The Sub-GNN provides significantly better localization granularity and achieves competitive classification performance, according to empirical evaluations on a synthetic dataset. This function-level approach lessens the workload associated with manual code auditing by enabling developers to more effectively detect and isolate problematic code.

We present a sizable, artificially produced dataset of Solidity contracts with function-level vulnerability annotations and structured semantic metadata to facilitate this fine-grained detection. This dataset has been carefully selected to guarantee label fidelity and injection realism, and it covers a wide range of vulnerability types, such as reentrancy, arithmetic errors, access control flaws, and more. By making it possible to train and assess detection models with fine localization capabilities, it closes a significant gap in the state of the art.

Simultaneously, this thesis addresses the interpretability issue by creat-

ing a pipeline for structured explanation generation driven by large language models (LLMs). We guide models like GPT-4o and refined CodeLlama-13B to produce human-readable vulnerability descriptions that include the bug name, affected area, description, impact assessment, and mitigation by combining synthetic data and CoT prompting. According to experimental findings, CoT prompting improves the generated explanations' coherence, informativeness, and clarity, which increases their value for developers. Furthermore, our automated evaluation pipeline using GPT-4o as a scoring agent confirms the effectiveness of our fine-tuned models across explanation quality dimensions including clarity, identification accuracy, and impact awareness.

These elements work together to create an end-to-end vulnerability analysis pipeline that combines semantic interpretation using LLMs with structural reasoning using Sub-GNN. By combining these two approaches, a more useful and developer-friendly smart contract auditing workflow is made possible, allowing for both precise bug detection and clear explanation.

This work paves the way for a number of exciting avenues for further investigation. The ability of the Sub-GNN to detect compositional and cross-contract vulnerabilities can be enhanced by adding support for inter-function and inter-contract reasoning. On the explanation side, usability could be further improved by utilizing LLMs to create interactive auditing assistants and integrating user feedback. Furthermore, we envision the potential integration of textual representations and graph-based embeddings into a single multimodal model for joint explanation and detection.

To sum up, this thesis advances the state of the art in smart contract vulnerability detection and explanation by introducing new models, datasets, and evaluation techniques. This work establishes the groundwork for blockchain systems that are more secure and interpretable by bridging the gap between automated classification and human-centered understanding.

# Bibliography


[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning Distributed Representations of Code, October 2018. arXiv:1803.09473 [cs].

[2] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection, August 2023. arXiv:2304.00409 [cs].

[3] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology*, 159:107221, July 2023.

[4] ConsenSys Diligence. Mythril: Symbolic-execution-based security analysis tool for evm bytecode. `https://github.com/ConsenSysDiligence/mythril`, 2025. Accessed: 2025-07-08.

[5] Crytic. Slither: Slither, the smart contract static analyzer. `https://github.com/crytic/slither`, 2025. Accessed: 2025-07-08.

[6] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541, June 2020. arXiv:1910.10601 [cs].

[7] Enzyme Finance (formerly Oyente). Oyente: An analysis tool for smart contracts. `https://github.com/enzymefinance/oyente`, 2025. Accessed: 2025-07-08.

[8] ETH SRI (Secure, Reliable, and Intelligent Systems Lab). Securify2: Security scanner for ethereum smart contracts supported by

the ethereum foundation and chainsecurity. `https://github.com/eth-sri/securify2`, 2025. Accessed: 2025-07-08.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages, September 2020. arXiv:2002.08155 [cs].

[10] Asem Ghaleb and Karthik Pattabiraman. How Effective are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, July 2020. arXiv:2005.11613 [cs].

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training Code Representations with Data Flow, September 2021. arXiv:2009.08366 [cs].

[12] Ayush Gurjar and B. R. Chandavarkar. Smart Contract Vulnerabilities and Detection Methods: A Survey. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7, Kamand, India, June 2024. IEEE.

[13] Gerardo Iuliano and Dario Di Nucci. Smart Contract Vulnerabilities, Tools, and Benchmarks: An Updated Systematic Literature Review, December 2024. arXiv:2412.01719 [cs].

[14] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks, February 2017. arXiv:1609.02907 [cs].

[15] Chen Liang, Qiang Wei, Jiang Du, Yisen Wang, and Zirui Jiang. Survey of source code vulnerability analysis based on deep learning. *Computers & Security*, 148:104098, January 2025.

[16] Christensen Choong Ming Jie, Daveena Sri A-P Michael Rajah, Gyvayhni A-P Ganisen, Tharanitharan A-L Chandran, Ahmad Sahban Rafsanjani, and Muhammed Basheer Jasser. A Secure Data Sharing Platform in the Era of Web 3.0 by Leveraging the Power of Blockchain Technology. In *2024 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*, pages 291–296, Shah Alam, Malaysia, June 2024. IEEE.

[17] Shaznin Sultana, Sadia Afreen, and Nasir U. Eisty. Code Vulnerability Detection: A Comparative Analysis of Emerging Large Language Models, September 2024. arXiv:2409.10490 [cs].

[18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks, February 2018. arXiv:1710.10903 [stat].

[19] Sally Junsong Wang, Jianan Yao, Kexin Pei, Hidedaki Takahashi, and Junfeng Yang. Detecting Buggy Contracts via Smart Testing, September 2024. arXiv:2409.04597 [cs].

[20] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Trans. Netw. Sci. Eng.*, 8(2):1133–1144, April 2021.

[21] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. CodeT5+: Open code large language models for code understanding and generation.

[22] Zhiyuan Wei, Jing Sun, Zijian Zhang, Xianhao Zhang, Xiaoxuan Yang, and Liehuang Zhu. Survey on Quality Assurance of Smart Contracts. *ACM Comput. Surv.*, 57(2):1–36, February 2025.

[23] Peiyan Zhang, Yuchen Yan, Xi Zhang, Chaozhuo Li, Senzhang Wang, Feiran Huang, and Sunghun Kim. TransGNN: Harnessing the Collaborative Power of Transformers and Graph Neural Networks for Recommender Systems, May 2024. arXiv:2308.14355 [cs].

[24] Jinyao Zhu, Xiaofei Xing, Guojun Wang, and Peiqiang Li. Opcode Sequences-Based Smart Contract Vulnerabilities Detection Using Deep Learning. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 284–291, Exeter, United Kingdom, November 2023. IEEE.

[25] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart Contract Vulnerability Detection using Graph Neural Network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 3283–3290, Yokohama, Japan, July 2020. International Joint Conferences on Artificial Intelligence Organization.

## 5.1 Publication

Improve Smart Contract Vulnerability Explanation with Synthetic Data and Chain-of-Thought Prompting (Accepted in NLDB 2025)