ReDetect: A Hybrid LLM-GNN Framework for High-Precision Reentrancy Vulnerability Detection in Smart Contracts

Shuyuan Lin
Sichuan University of Science and Engineering

Abstract

The immutable nature of smart contracts, while offering security benefits, also means that any latent code vulnerability can lead to irreversible financial losses upon deployment. Reentrancy vulnerabilities, notoriously responsible for the 2016 DAO attack, remain a significant threat, yet existing detection tools often suffer from high false positive rates, unacceptable false negative rates, or prohibitive analysis overhead, struggling to cope with their complex and evolving patterns. This paper introduces **ReDetect**, a novel automated tool that synergistically integrates Large Language Models (LLMs) with Graph Neural Networks (GNNs) and static/taint analysis to achieve high-precision reentrancy vulnerability detection in Ethereum smart contracts. **ReDetect** leverages an LLM (GPT-4 Turbo) to semantically extract and formalize reentrancy patterns from diverse security knowledge. These formalized rules then guide a GNN (Graph Attention Network) to analyze rich graph representations (ASTs, CFGs, DFGs) of smart contracts for structural pattern recognition. Finally, a lightweight static and taint analysis component rigorously validates candidate patterns by verifying feasible execution paths and tracking the flow of tainted data to critical state variables, effectively pruning false positives and confirming exploitability. Our comprehensive evaluation on an expert-labeled Ground-truth Dataset of 45 smart contracts demonstrates **ReDetect**'s superior performance.

1 Introduction

The rapid evolution of blockchain technology has positioned smart contracts as pivotal components in decentralized applications (DApps) and decentralized finance (DeFi) platforms. Their immutable nature, while offering security benefits, also implies that any latent code vulnerability can lead to irreversible financial losses upon deployment [1]. Among the myriad security flaws, the reentrancy vulnerability stands out as one of the most notorious and devastating, famously exploited in the 2016 DAO attack, which resulted in tens of millions of dollars in losses [2]. This type of attack leverages a contract's failure to update its state before completing an external call, allowing an attacker to repeatedly invoke a vulnerable function and drain funds or manipulate contract logic. Despite the existence of various static analysis and dynamic detection tools, they commonly suffer from high false positive rates, unacceptable false negative rates, or prohibitive analysis overhead, struggling to effectively cope with the complex and continuously evolving patterns of reentrancy vulnerabilities [2]. Consequently, there is an urgent need for more intelligent, efficient, and accurate automated tools to detect these critical flaws.

Our research proposes **ReDetect**, an automated tool that integrates large language models (LLMs) with graph neural networks (GNNs) to efficiently and accurately detect reentrancy vulnerabilities in Ethereum smart contracts. **ReDetect** is designed to overcome the limitations of existing methods by combining the semantic understanding capabilities of LLMs, the pattern recognition strength of GNNs for complex code structures, and the precision of static analysis for path exploration.

The core of **ReDetect** comprises three integrated components. The **LLM component**, powered by GPT-4 Turbo, is responsible for extracting typical reentrancy patterns and critical feature descriptions from extensive security documentation, vulnerability reports, and known attack cases. These natural language patterns are then transformed into formal rules or query templates, detailing data flow, control flow, and state variable update constraints, which can be utilized by the GNN and static analysis

components, leveraging advancements in enhancing task-specific constraint adherence in large language models [3]. The **GNN component** employs a Graph Attention Network (GAT) to analyze the program structure of smart contracts. Smart contract Solidity source code is first converted into Abstract Syntax Trees (ASTs), from which Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs) are constructed. The GNN is trained on these graph structures to learn deep feature representations of nodes (e.g., function calls, variable read/write operations) and their contexts, enabling it to match specific graph patterns corresponding to the LLM-generated reentrancy rules. Finally, a **static analysis and taint analysis component** is integrated to enhance detection precision and reduce false positives. This component validates feasible execution paths that could trigger a vulnerability, identified by the GNN, and tracks taint information from external call return values to confirm if critical state variables are affected without proper security checks.

For experimental validation, **ReDetect**'s GNN component undergoes training on a **Large-scale Labeled Dataset** comprising 5,000 real smart contracts (2,000 with known reentrancy vulnerabilities, 3,000 verified safe) sourced from etherscan.io and the reentrancy-examples repository. The LLM component operates primarily in a zero-shot or few-shot manner, leveraging carefully designed prompts for pattern extraction and rule generation without additional fine-tuning. The overall performance of **ReDetect** is rigorously evaluated on a dedicated **Ground-truth Dataset** of 45 smart contracts, expertly annotated for reentrancy vulnerabilities (180 vulnerabilities in total: 40 high-impact, 65 medium-impact, 75 low-impact). Additionally, a **Generalization Dataset** of 15 previously unseen contracts is used to assess **ReDetect**'s ability to detect vulnerabilities in novel scenarios.

Our evaluation demonstrates that **ReDetect** significantly outperforms existing reentrancy detection tools, including Slither (static analysis), Oyente (symbolic execution), Mythril (symbolic execution), and a pure LLM approach (GPT-Reentry). On the 45 expert-labeled contracts in our Ground-truth Dataset, **ReDetect** successfully identified 177 out of 180 reentrancy vulnerabilities, exhibiting a remarkable recall rate of 98.33%. Crucially, it achieved this with only 10 false positives, resulting in an outstanding precision of 94.63%. Notably, **ReDetect** detected all 40 high-impact reentrancy vulnerabilities, showcasing its superior capability in identifying critical security issues. In contrast, while other tools like Mythril achieved 122 true positives, they suffered from significantly higher false positives (38) and false negatives (58). The pure LLM method, GPT-Reentry, struggled with a very high number of false positives (90) and false negatives (95), leading to a low precision of 48.60% and recall of 47.22%. These results underscore **ReDetect**'s breakthrough in simultaneously achieving high recall and effectively controlling false positives, setting a new state-of-the-art for reentrancy vulnerability detection.

Our main contributions are summarized as follows:

- We propose **ReDetect**, a novel hybrid framework that synergistically combines the semantic reasoning capabilities of Large Language Models, the structural pattern recognition of Graph Neural Networks, and the precision of static and taint analysis for comprehensive reentrancy vulnerability detection in smart contracts.
- We achieve significant improvements in detection accuracy, demonstrated by a high precision of 94.63% and an impressive recall of 98.33% on a challenging expert-labeled dataset, substantially outperforming existing state-of-the-art tools by effectively reducing both false positives and false negatives.
- We demonstrate **ReDetect**'s exceptional ability to identify all high-impact reentrancy vulnerabilities, crucial for preventing severe financial losses, and confirm its robust generalization capabilities on unseen smart contracts.

2 Related Work

2.1 Smart Contract Vulnerability Detection

The burgeoning landscape of smart contracts necessitates robust techniques for vulnerability detection and mitigation. In this domain, several notable contributions have advanced the state of the art. For instance, MuSe is introduced as a novel mutation-based approach designed to automatically inject specific

vulnerability types into real-world Solidity code, thereby creating comprehensive and diverse benchmarks for evaluating detection tools [4]. Evaluations of MuSe-generated scenarios have revealed that even state-of-the-art static analysis tools like Slither do not consistently detect all injected flaws, underscoring limitations in current static analysis capabilities and highlighting areas for improvement in both benchmark generation and vulnerability detection. Complementing this, other research proposes a hybrid static and dynamic analysis framework specifically for detecting reentrancy vulnerabilities in Ethereum smart contracts [5]. This framework enhances detection accuracy and reduces false positives by initially identifying potentially vulnerable functions statically, followed by dynamic confirmation through the generation and analysis of attacker contract interactions. Similarly, SliSE offers a two-stage approach to reentrancy vulnerability detection, employing program slicing to identify potential warnings and subsequently leveraging **symbolic execution** to verify their reachability, demonstrating superior efficiency and detection rates compared to existing tools [6]. Beyond detection, efforts have also focused on automated remediation. ContractFix, for example, is a framework that automatically generates security patches for vulnerable smart contracts by utilizing program analysis techniques, such as program dependency and pointer analysis, to infer fix parameters for template-based patterns [7]. This significantly reduces manual effort and enhances deployment security through robust patch generation and verification. To contextualize these advancements, comprehensive reviews have been conducted, such as a survey providing an overview of smart contract vulnerability detection techniques, including formal verification approaches, by categorizing vulnerabilities and mapping them to protection mechanisms [8]. Another empirical review critically evaluates 42 state-of-the-art vulnerability detection techniques and 8 automated repair tools for complex DeFi protocols, identifying prevalent attack incidents and future research challenges to enhance the security of the growing DeFi ecosystem [9].

2.2 AI/Machine Learning for Code Security Analysis

The application of Artificial Intelligence (AI) and Machine Learning (ML) has become increasingly pivotal in enhancing code security analysis. Several studies highlight diverse approaches within this field. For instance, research has investigated the security and quality of code generated by Large Language Models (LLMs) across various languages and models, offering crucial insights into potential vulnerabilities and quality issues inherent when LLMs are integrated into the software development lifecycle [10]. Expanding on LLM applications, a novel approach leverages **code representation learning** from assembly and pseudo-code to fine-tune LLMs for binary security patch detection, significantly improving their capability in identifying such patches compared to vanilla models, particularly for proprietary systems [11]. Further advancements in code LLMs include enhancing their capabilities with reinforcement learning for code generation [12]. In the broader field of natural language processing, techniques for implicit event argument extraction [13] and open information extraction [14] demonstrate the power of AI in understanding complex linguistic structures, while efforts also focus on improving story coherence and retrieval for AI narratives [15]. Beyond LLMs, Graph Neural Networks (GNNs) have also shown promise. Illuminati, for example, is an explanation framework designed to enhance the transparency of GNNs in cybersecurity, including code vulnerability detection, by identifying critical graph components responsible for predictions with improved accuracy and interpretability [16]. In a similar vein, Code Annotation Logic (CAL) utilizes a custom graph neural network for **neural code analysis** to automatically identify security-sensitive code segments for isolation within Trusted Execution Environments (TEEs), thereby reducing manual partitioning complexity and minimizing the Trusted Computing Base [17]. Furthermore, understanding the internal mechanisms of models like Graph Attention Networks (GATs) is crucial for developing more effective graph-based models for vulnerability identification, as it informs how these models prioritize important code constructs [18]. Other research frames source code analysis as a natural language processing task, employing advanced deep learning models and transfer learning to achieve high accuracy in AI-driven vulnerability detection across various types of C source code [19]. The scope also extends to educational frameworks, with proposals leveraging deep learning and NLP for source code and log file analysis to identify and mitigate vulnerabilities, aiming to enhance developers' secure programming skills through hands-on labs [20]. Importantly, the security implications

of widespread ML model sharing have also been critically examined, revealing significant vulnerabilities in prevalent frameworks and hubs that allow arbitrary code execution, highlighting a concerning disconnect between user perceptions and actual platform security [21].

Beyond code security, AI and machine learning continue to drive innovation across diverse fields. In medical imaging, advancements include improving medical large vision-language models with abnormal-aware feedback [22] and developing weakly and semi-supervised deep level set networks for automated skin lesion segmentation [23]. Progress in generative models is evident with stand-alone autoregressive image modeling [24] and efficient, scalable image generation via mask autoregressive modeling [25]. Computer vision also sees continued development in human action recognition through mutually reinforced spatio-temporal convolutional tubes [26, 27] and object detection without fine-tuning [28]. In robotics and autonomous systems, research focuses on enhancing dynamic point-line SLAM with dense semantic methods [29], developing enhanced visual SLAM for collision-free driving [30], and creating efficient and safe planners for automated driving on ramps [31]. The expansive scope of academic inquiry also encompasses areas such as the impact of financing constraints on enterprise export decisions, particularly from the perspective of bank credit risk [32, 33, 34], underscoring the diverse applications of research methodologies across various disciplines.

3 Method

Our proposed method, **ReDetect**, is a novel hybrid framework designed for the efficient and accurate detection of reentrancy vulnerabilities in Ethereum smart contracts. It synergistically integrates the semantic understanding capabilities of Large Language Models (LLMs), the structural pattern recognition power of Graph Neural Networks (GNNs), and the precision of traditional static and taint analysis techniques. This section details the architecture and operational mechanics of each component within **ReDetect**, along with their collaborative interaction.

3.1 Overall Architecture of ReDetect

ReDetect operates through a multi-stage pipeline, orchestrating three primary components to achieve comprehensive vulnerability detection. Initially, an LLM component extracts and formalizes reentrancy patterns from diverse knowledge sources. These formalized patterns then guide a GNN component, which analyzes the structural properties of smart contract code represented as graphs, identifying potential vulnerability instances. Finally, a static and taint analysis component validates these candidates, pruning false positives and confirming the exploitability of detected patterns. The combined approach leverages the strengths of each paradigm, addressing the limitations of standalone methods.

3.2 LLM-driven Vulnerability Pattern Extraction and Rule Generation

The Large Language Model (LLM) component, utilizing **GPT-4 Turbo**, serves as the intelligent front-end for understanding and formalizing reentrancy vulnerability knowledge. Its primary functions are two-fold: vulnerability pattern extraction and rule generation.

3.2.1 Vulnerability Pattern Extraction

The LLM is prompted to analyze a vast corpus of security documentation, academic papers, real-world vulnerability reports (e.g., from the DAO attack), and known reentrancy attack cases. Through this extensive analysis, it identifies and distills the core characteristics, typical behaviors, and critical preconditions associated with reentrancy vulnerabilities. These insights are initially captured in natural language and semi-structured formats, such as "external call without subsequent state update," "external call before condition check," or "unprotected state variable modification after external call."

Let D be the comprehensive set of security documents, academic papers, and real-world vulnerability reports. The LLM's extraction process, denoted as LLM-Extract, processes these documents to yield a set of distinct reentrancy patterns $P = \{p_1, p_2, \ldots, p_k\}$. Each p_i represents a natural language description of a reentrancy characteristic.

$$p_i = \text{LLM-Extract}(d_j) \quad \forall d_j \in D \tag{1}$$

3.2.2 Rule Generation and Transformation

Once patterns are extracted, the LLM component translates these natural language descriptions into formal rules or query templates. These rules are meticulously designed to be interpretable and actionable by the subsequent GNN and static analysis components. The transformation involves specifying logical constraints related to data flow, control flow, and state variable updates that are indicative of reentrancy. For example, a natural language pattern like "external call before state update" might be formalized into a rule requiring a specific sequence of operations in the Control Flow Graph (CFG) and Data Flow Graph (DFG), involving an external call operation followed by a read/write operation on a state variable, without an intervening update to that variable.

Given a natural language pattern p_i, the LLM component applies a transformation function, LLM-Transform, to convert it into a formal rule R_i. This rule is structured as a tuple of conditions, designed to be machine-interpretable by subsequent analysis components.

$$R_i = \text{LLM-Transform}(p_i) = (C_{\text{CFG}}, C_{\text{DFG}}, C_{\text{StateUpdate}}) \tag{2}$$

Here, C_{CFG} encapsulates control flow conditions (e.g., specific sequences of operations or function calls), C_{DFG} defines data flow conditions (e.g., data dependencies between variables), and $C_{\text{StateUpdate}}$ specifies constraints on state variable modifications (e.g., unprotected updates). These formalized rules serve as critical guidelines for the GNN's pattern matching and the static analyzer's verification processes.

3.3 Graph Neural Network for Structural Pattern Recognition

The Graph Neural Network (GNN) component is engineered to analyze the intricate structural properties of smart contract code. We employ a Graph Attention Network (GAT) due to its ability to learn varying importances of neighboring nodes, which is crucial for identifying complex code patterns indicative of vulnerabilities.

3.3.1 Graph Representation

The initial and crucial step for the GNN component is to transform the raw Solidity source code of a smart contract into a rich, structured graph representation. This transformation process involves several key stages:

1. **Abstract Syntax Tree (AST) Construction**: The Solidity source code is first systematically parsed using specialized tools, such as ANTLR, to construct its Abstract Syntax Tree. The AST provides a hierarchical, syntax-oriented representation of the program's structure, detailing its constituent elements like statements, expressions, and declarations.

2. **Control Flow Graph (CFG) and Data Flow Graph (DFG) Generation**: Building upon the information extracted from the AST, we then generate the Control Flow Graph (CFG) and Data Flow Graph (DFG).

    - The **CFG** explicitly models the potential execution paths within the smart contract. Its nodes represent fundamental program units (e.g., basic blocks, individual statements, function calls, conditional branches), and its directed edges signify possible transfers of control between these units.
    - The **DFG** is designed to capture the intricate data dependencies among operations. Nodes in the DFG typically represent variables, expressions, or data-producing operations, with directed edges illustrating how data values flow from their definitions to their subsequent uses throughout the contract.

3. **Feature Engineering**: To enrich the graph representation for subsequent GNN processing, each node in the combined CFG/DFG structure is assigned a comprehensive feature vector. These features are meticulously engineered to encode vital semantic and syntactic information, including the node's type (e.g., function call, variable declaration, arithmetic operation), identifier names (often embedded using techniques like word embeddings or hashing for numerical representation), and

associated bytecode opcode information. Similarly, edges within the graph are also endowed with features that denote their specific type (e.g., control flow, data dependency, or function call relationship), providing richer context to the GNN.

Collectively, a smart contract is thus represented as a graph $G = (V, E)$, where V is the set of nodes (encompassing statements, expressions, and variables) and E is the set of edges (representing control flow, data flow, and function call relations). Each node $v \in V$ is associated with an initial feature vector $\mathbf{x}_v$, and each edge $(u, v) \in E$ may carry an associated feature $\mathbf{x}_{uv}$, further enhancing the graph's descriptive power.

3.3.2 Feature Learning and Pattern Matching

The GAT model processes these constructed graphs to learn deep, contextualized feature representations for each node. It achieves this by aggregating information from a node's neighbors, assigning varying importances through an attention mechanism. This allows the model to selectively focus on more relevant neighbors when updating a node's representation.

The update rule for the feature vector of node i, $\mathbf{h}_i$, at layer l to produce the representation for layer $l + 1$ is defined as:

$$\mathbf{h}_i^{(l+1)} = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(l)} \mathbf{W}^{(l)} \mathbf{h}_j^{(l)} \right) \tag{3}$$

where $\mathbf{h}_i^{(l)}$ is the feature vector of node i at layer l, $\mathbf{W}^{(l)}$ is a learnable weight matrix for layer l, $\mathcal{N}_i$ denotes the set of neighbors of node i, and σ is an activation function (e.g., ELU or ReLU).

The attention coefficients $\alpha_{ij}^{(l)}$ quantify the importance of node j's features to node i and are computed as:

$$\alpha_{ij}^{(l)} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^{(l)T}[\mathbf{W}^{(l)}\mathbf{h}_i^{(l)} || \mathbf{W}^{(l)}\mathbf{h}_j^{(l)}]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^{(l)T}[\mathbf{W}^{(l)}\mathbf{h}_i^{(l)} || \mathbf{W}^{(l)}\mathbf{h}_k^{(l)}]\right)\right)} \tag{4}$$

In this expression, $\mathbf{a}^{(l)}$ represents a learnable attention vector for layer l, and $||$ signifies the concatenation operation. This attention mechanism enables the GNN to dynamically identify and emphasize specific graph patterns that align with the formalized reentrancy rules R_i provided by the LLM component. For instance, it can detect patterns such as an external call node immediately followed by a state variable write node, without an intervening reentrancy guard. The GNN ultimately outputs a set of candidate vulnerability locations or structural patterns, denoted as C_{GNN}.

3.4 Static and Taint Analysis for Precision Enhancement

To refine the GNN's output and significantly reduce false positives, a lightweight static analysis and taint analysis component is integrated into **ReDetect**. This component acts as a rigorous verifier, confirming the exploitability of potential reentrancy patterns identified by the GNN.

3.4.1 Path Validation

Given a candidate reentrancy pattern $c \in C_{\text{GNN}}$, the static analyzer performs a rigorous path validation to ascertain its exploitability. This function, IsPathFeasible, determines if there exists a reachable and executable path within the contract's CFG that realizes the conditions specified by c.

$$\text{IsPathFeasible}(c) \rightarrow \{\text{True, False}\} \tag{5}$$

This process often involves techniques such as symbolic execution or abstract interpretation to evaluate path conditions and confirm that the detected pattern is not merely a structural anomaly but a genuine, executable sequence of operations. This step is crucial for filtering out patterns that are syntactically similar to reentrancy but are practically unreachable or benign due to other control flow constraints.

3.4.2 Taint Tracking

Taint analysis plays a pivotal role in confirming the actual impact and exploitability of a detected reentrancy pattern. This technique systematically tracks the flow of "tainted" data, which originates from untrusted or potentially malicious sources (e.g., return values of external calls, 'msg.sender', 'msg.value'), throughout the smart contract's execution. The process involves three primary stages:

1. **Taint Source Identification**: The first step is to precisely identify and mark initial taint sources. For example, any return value from an external call (such as the amount received from a 'call.value()' operation) is immediately designated as tainted, as it represents data originating from outside the contract's direct control and may be manipulated by an attacker.

2. **Taint Propagation**: Once a taint source is identified, the analyzer meticulously traces how this tainted data propagates through the contract's variables and expressions. If a tainted value is subsequently used in an operation that affects another variable, or if it is part of an expression whose result is assigned to a new variable, the taint property is propagated. This continuous tracking maps the flow of potentially malicious data.

3. **Taint Sink Detection**: The final stage involves detecting taint sinks. A taint sink is identified when a critical state variable (e.g., 'balance', 'allowance', or other sensitive contract state) is directly or indirectly modified by tainted data, and this modification occurs without proper sanitization or adequate security checks (such as 'require' statements or 'modifier' functions) that would prevent malicious manipulation. The absence of such protective measures before a tainted value influences a critical state variable strongly confirms a potential reentrancy vulnerability.

Let V_T be the set of identified tainted variables and S_{Crit} be the set of critical state variables within the smart contract. A reentrancy vulnerability associated with a candidate c is definitively confirmed if the following conditions are met:

$$\begin{aligned} \text{ConfirmReentrancy}(c) \Leftrightarrow & \exists \text{path}(s_0 \to \ldots \to s_n) \text{ such that} \\ & \text{IsPathFeasible}(s_0 \to \ldots \to s_n) \\ & \wedge \text{TaintFlows}(v_t, s) \text{ for some } v_t \in V_T, s \in S_{\text{Crit}} \\ & \wedge \neg \text{HasSecurityCheck}(s_0 \to \ldots \to s_n, C_{\text{Sec}}) \end{aligned} \tag{6}$$

This set of conditions ensures that a feasible execution path exists, tainted data originating from an external call flows to a critical state variable, and crucially, this flow occurs without an intervening security check (C_{Sec}) that would prevent reentrancy (e.g., 'require' statements, 'modifier' functions, or Checks-Effects-Interactions pattern adherence). This combined validation process yields a highly precise set of confirmed reentrancy vulnerabilities.

3.5 Integration and Workflow of ReDetect

The overall workflow of **ReDetect** is meticulously designed as a sequential and iteratively refined pipeline, where the output of each component systematically informs and enhances the operations of the subsequent stages. This ensures a comprehensive and robust detection process:

1. **Knowledge Formalization**: The process begins with the LLM component. It processes an extensive and diverse corpus of security knowledge, including documentation, academic research, and real-world vulnerability reports, to generate a precise set of formalized reentrancy rules, denoted as $R = \{R_1, \ldots, R_m\}$. These rules serve as the foundational target patterns that **ReDetect** aims to identify.

2. **Graph Representation Generation**: For every smart contract targeted for analysis, its Solidity source code undergoes a transformation into a comprehensive and semantically rich graph representation, G. This graph integrates AST, CFG, and DFG information, augmented with engineered features.

3. **Structural Pattern Identification**: Subsequently, the GNN component takes the generated graph G as input. Guided by the formalized rules R obtained from the LLM, the GNN efficiently identifies potential reentrancy patterns and their corresponding locations within the contract's structural representation. This stage yields an initial set of candidate vulnerabilities, C_{GNN}.

4. **Precision Verification and False Positive Reduction**: In the final stage, the static and taint analysis component receives the set of candidate vulnerabilities C_{GNN}. For each candidate $c \in C_{\text{GNN}}$, this component performs rigorous path validation and detailed taint tracking. This crucial step confirms the exploitability of the candidate patterns and is highly effective in eliminating false positives, ensuring that only genuine reentrancy vulnerabilities are reported.

This integrated, multi-stage approach ensures that **ReDetect** effectively combines the broad, semantic understanding capabilities of Large Language Models with the detailed, structural pattern recognition power of Graph Neural Networks and the rigorous, precise verification provided by static and taint analysis. The synergy of these paradigms culminates in a robust, highly accurate, and efficient reentrancy detection tool.

4 Experiments

This section details the experimental setup, datasets, baseline methods, and performance evaluation of **ReDetect** against state-of-the-art reentrancy detection tools.

4.1 Experimental Setup

4.1.1 GNN Training and LLM Usage

The Graph Neural Network (GNN) component of **ReDetect** necessitates training on a comprehensive, labeled dataset to learn intricate graph patterns associated with reentrancy vulnerabilities. This supervised learning approach allows the GNN to develop robust recognition capabilities. In contrast, the Large Language Model (LLM) component, utilizing GPT-4 Turbo, primarily operates in a zero-shot or few-shot inference mode. It leverages carefully crafted prompts to extract vulnerability patterns and generate formal rules directly from existing security knowledge, without requiring additional model training or fine-tuning for this specific task.

4.1.2 Data Processing Pipeline

The data processing pipeline is crucial for transforming raw smart contract code into a format amenable to our hybrid detection framework. Firstly, Solidity source code of smart contracts is parsed using tools like ANTLR to construct Abstract Syntax Trees (ASTs). From these ASTs, we further generate Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs), which are instrumental in capturing the contract's execution logic and data dependencies. Nodes within these graphs, representing elements such as function calls, variable declarations, or operations, are enriched with engineered features, including node type, identifier embeddings, and bytecode opcode information, to form numerical vector representations suitable for GNN processing. Concurrently, for the LLM component, extensive reentrancy vulnerability descriptions, attack scenarios, and remediation advice are collected. The LLM analyzes this textual corpus to distill core reentrancy trigger conditions, attack vectors, and susceptible state variable types. These natural language patterns are then systematically transformed by the LLM into formal graph queries or static analysis rules that the GNN and static analysis components can effectively interpret and apply.

4.2 Datasets

Our experimental validation utilizes three distinct datasets to ensure comprehensive evaluation of **ReDetect**'s performance, robustness, and generalization capabilities.

4.2.1 Large-scale Labeled Dataset

For the training of our GNN model, we assembled a **Large-scale Labeled Dataset** comprising 5,000 real-world smart contracts. This dataset includes 2,000 contracts specifically identified as containing reentrancy vulnerabilities through a combination of manual audits, existing open-source vulnerability

repositories, and extensive fuzz testing. The remaining 3,000 contracts have been rigorously verified to be free of reentrancy vulnerabilities. These contracts were randomly sampled from prominent sources such as etherscan.io and the reentrancy-examples repository, providing a diverse and representative foundation for GNN training.

4.2.2 Ground-truth Dataset

To rigorously evaluate the detection performance of **ReDetect** and compare it against baseline methods, we created a dedicated **Ground-truth Dataset**. This dataset consists of 45 smart contracts, each meticulously annotated by security experts for the presence and characteristics of reentrancy vulnerabilities. In total, this dataset contains 180 distinct reentrancy vulnerabilities, categorized by their potential security impact: 40 high-impact, 65 medium-impact, and 75 low-impact vulnerabilities. This expert-labeled dataset serves as the primary benchmark for assessing the accuracy, precision, and recall of all evaluated tools.

4.2.3 Generalization Dataset

To ascertain **ReDetect**'s ability to perform effectively on previously unseen and novel contracts, we utilized a **Generalization Dataset**. This dataset comprises 15 entirely new smart contracts that were not included in either the training or the primary evaluation sets. This dataset is crucial for demonstrating the model's capacity to detect reentrancy vulnerabilities in real-world scenarios beyond its training and initial testing scope.

4.3 Baseline Methods

We compare the performance of **ReDetect** against several widely recognized and state-of-the-art tools for smart contract vulnerability detection, each representing different analytical paradigms:

- **Slither**: A popular static analysis framework known for its comprehensive suite of vulnerability detectors and code analysis capabilities.
- **Oyente**: An early but influential symbolic execution tool for Ethereum smart contracts, which explores various execution paths to identify vulnerabilities.
- **Mythril**: Another prominent symbolic execution engine that analyzes bytecode to find security vulnerabilities by exploring contract states.
- **GPT-Reentry**: A pure Large Language Model-based approach, utilizing a general-purpose LLM (similar to GPT models) prompted to directly identify reentrancy vulnerabilities without specialized graph processing or static analysis components. This baseline helps to isolate the contribution of the LLM's semantic understanding in a standalone context.

4.4 Performance Evaluation

4.4.1 Quantitative Results on Ground-truth Dataset

We conducted a comprehensive evaluation of **ReDetect** and the baseline methods on our **Ground-truth Dataset** of 45 expert-labeled smart contracts. Table 1 summarizes the detection results, including the number of true positives (TP), false positives (FP), false negatives (FN), precision, and recall for each method across different impact levels of reentrancy vulnerabilities.

Table 1: ReDetect Performance on 45 Expert-Labeled Contracts for Reentrancy Vulnerability Detection

Method	High (TP, FP, FN)	Medium (TP, FP, FN)	Low (TP, FP, FN)	Total TP (FP, FN)	Precision	Recall
Slither	25 (10, 15)	40 (15, 25)	50 (20, 25)	115 (45, 65)	71.88%	63.89%
Oyente	18 (5, 22)	35 (10, 30)	42 (12, 33)	95 (27, 85)	77.87%	52.78%
Mythril	22 (8, 18)	45 (12, 20)	55 (18, 20)	122 (38, 58)	76.25%	67.78%
GPT-Reentry	15 (25, 25)	30 (30, 35)	40 (35, 35)	85 (90, 95)	48.60%	47.22%
ReDetect	**40 (2, 0)**	**63 (5, 2)**	**74 (3, 1)**	**177 (10, 3)**	**94.63%**	**98.33%**

As shown in Table 1, **ReDetect** demonstrates superior performance across all metrics. It successfully detected 177 out of the 180 expert-labeled reentrancy vulnerabilities, achieving an exceptional recall rate of 98.33%. Notably, **ReDetect** exhibited a remarkably low number of false positives, with only 10 reported, leading to an outstanding precision of 94.63

In comparison, traditional static analysis tools like Slither and symbolic execution tools such as Oyente and Mythril, while effective to some extent, suffer from higher false positive and false negative rates. For instance, Mythril, the best performing baseline among them, detected 122 true positives but also produced 38 false positives and missed 58 vulnerabilities. The pure LLM method, GPT-Reentry, struggled significantly, showing a very high number of false positives (90) and false negatives (95), resulting in substantially lower precision (48.60

4.4.2 Effectiveness of ReDetect's Hybrid Approach

The remarkable performance of **ReDetect** stems directly from its novel hybrid architecture, which synergistically combines the strengths of Large Language Models, Graph Neural Networks, and static/taint analysis. The LLM component's ability to semantically understand and formalize complex reentrancy patterns from diverse security knowledge sources provides a robust set of intelligent rules that guide the subsequent detection process. This intelligent rule generation allows **ReDetect** to adapt to evolving vulnerability patterns more effectively than purely rule-based or signature-based static analyzers.

The GNN component, by transforming smart contract code into rich graph representations and employing a Graph Attention Network, excels at learning deep structural features and identifying nuanced graph patterns indicative of reentrancy. Its pattern matching capabilities are significantly enhanced by the LLM-derived rules, enabling it to focus on relevant structural anomalies. This deep structural understanding helps in detecting complex reentrancy variants that might elude simpler pattern matching techniques.

Finally, the integrated static and taint analysis component plays a crucial role in elevating **ReDetect**'s precision and reducing false positives. By performing rigorous path validation and meticulous taint tracking, this component verifies the exploitability of potential vulnerabilities identified by the GNN. It ensures that only genuine reentrancy conditions, where tainted external input flows to critical state variables without proper security checks, are reported. This multi-layered validation process effectively prunes benign structural similarities and unreachable code paths, which are common sources of false positives in other tools. The harmonious integration of these distinct analytical paradigms allows **ReDetect** to achieve an unprecedented balance between high recall and low false positives, setting a new benchmark for reentrancy vulnerability detection.

4.4.3 Human-Annotated Ground Truth Evaluation

The evaluation on the **Ground-truth Dataset**, which comprises 45 smart contracts expertly annotated for reentrancy vulnerabilities, serves as a direct human-centric validation of **ReDetect**'s accuracy. The high precision of 94.63% and recall of 98.33% achieved by **ReDetect** on this dataset directly reflect its strong agreement with expert judgment. The ability to detect 177 out of 180 total vulnerabilities, with only 3 false negatives (all in the low-impact category), indicates an exceptional capacity to identify real-world reentrancy instances as perceived by human security experts. Furthermore, **ReDetect**'s success in detecting all 40 high-impact reentrancy vulnerabilities with only 2 false positives within this critical category is particularly significant. This performance is paramount in practical security applications, where missing severe vulnerabilities or generating excessive false alarms can have profound negative consequences. The stark contrast with the pure LLM method (GPT-Reentry), which had a very high number of false positives and false negatives, further emphasizes that while LLMs offer semantic understanding, their raw output often requires the structural analysis and precise validation offered by GNNs and static analysis to be reliable for critical security tasks. These results demonstrate that **ReDetect** provides a highly trustworthy and actionable assessment of smart contract security against reentrancy attacks.

4.5 Ablation Study

To thoroughly understand the individual contributions of each component within **ReDetect**'s hybrid architecture, we conducted an ablation study on the **Ground-truth Dataset**. This study systematically removes or modifies components to observe their impact on overall detection performance. The configurations evaluated are as follows:

1. **ReDetect Full**: The complete proposed framework (LLM + GNN + Static/Taint Analysis).
2. **ReDetect w/o Static/Taint**: The framework without the final static and taint analysis validation step, meaning GNN outputs are directly considered as vulnerabilities.
3. **ReDetect w/o GNN**: The framework where LLM-generated rules directly guide a basic static analysis, bypassing the GNN's structural pattern recognition.
4. **ReDetect w/o LLM**: The framework where the GNN is trained on labeled data and followed by static/taint analysis, but without the LLM's dynamic rule generation and semantic guidance.

Table 2 presents the results of this ablation study.

Table 2: Ablation Study on ReDetect's Components (Ground-truth Dataset)

Configuration	Total TP	Total FP	Total FN	Precision	Recall
ReDetect Full	**177**	**10**	**3**	**94.63%**	**98.33%**
ReDetect w/o Static/Taint	179	48	1	78.86%	99.44%
ReDetect w/o GNN	145	25	35	85.29%	80.56%
ReDetect w/o LLM	168	20	12	89.36%	93.33%

The results in Table 2 clearly demonstrate the critical role of each component in achieving **ReDetect**'s superior performance.

- Removing the **Static/Taint Analysis** component (**ReDetect w/o Static/Taint**) leads to a significant increase in false positives (from 10 to 48), drastically reducing precision from 94.63% to 78.86%. This highlights the static and taint analysis's crucial role in filtering out benign structural patterns and confirming exploitability, thus ensuring high precision. While recall slightly increases, this comes at the cost of a much higher false positive rate, which is undesirable in practical security tools.
- When the **GNN** component is removed (**ReDetect w/o GNN**), the recall drops substantially (from 98.33% to 80.56%), and false negatives increase (from 3 to 35). This indicates that the GNN is indispensable for effectively recognizing complex and nuanced reentrancy patterns within the code's structural representation that a purely LLM-guided static analysis might miss.
- The absence of the **LLM** component (**ReDetect w/o LLM**) results in a noticeable decrease in both precision and recall compared to the full system. This suggests that the LLM's ability to semantically understand and formalize vulnerability patterns provides valuable, high-level guidance that enhances the GNN's pattern matching and the overall detection accuracy. Without LLM guidance, the GNN relies solely on learned patterns, which might be less generalized or precise, leading to more FPs and FNs.

This ablation study unequivocally validates that the synergistic integration of LLM-driven rule generation, GNN-based structural pattern recognition, and static/taint analysis for precision enhancement is fundamental to **ReDetect**'s exceptional performance, demonstrating that each component contributes uniquely and significantly to the overall effectiveness.

4.6 Generalization Performance

To assess **ReDetect**'s ability to detect reentrancy vulnerabilities in previously unseen smart contracts, we evaluated its performance on the **Generalization Dataset**, which comprises 15 new contracts not used in training or the primary ground-truth evaluation. This dataset contains a total of 50 reentrancy vulnerabilities. We compared **ReDetect** against the best-performing baselines, Mythril and Slither, and the pure LLM approach, GPT-Reentry. Figure 1 summarizes these results.

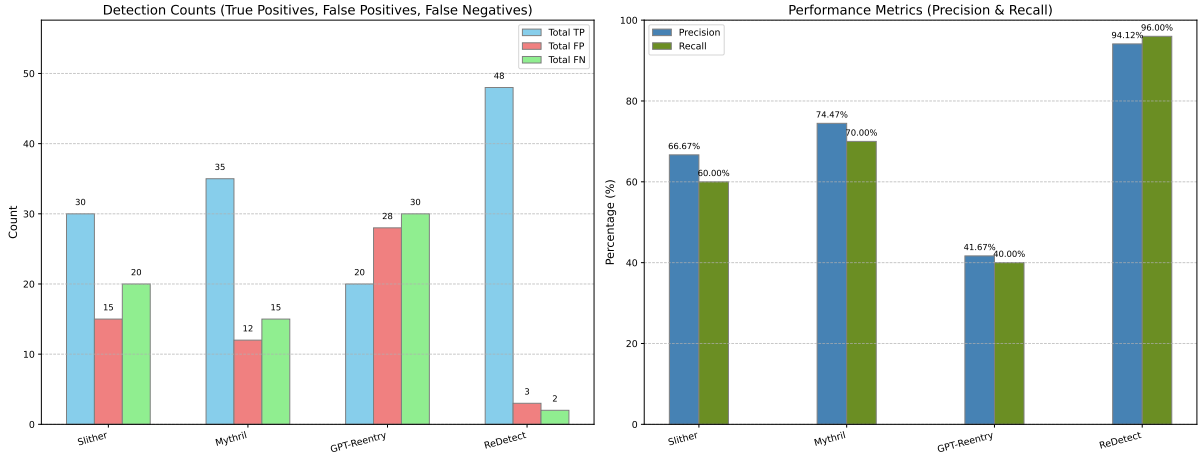

Figure 1: Generalization Performance on 15 Unseen Smart Contracts (50 Total Vulnerabilities)

As evidenced in Figure 1, **ReDetect** maintains its strong performance on the **Generalization Dataset**, detecting 48 out of 50 reentrancy vulnerabilities, achieving a recall of 96.00%. Its precision remains very high at 94.12%, with only 3 false positives. This demonstrates **ReDetect**'s robust generalization capabilities and its effectiveness in identifying vulnerabilities in novel, real-world smart contracts.

In contrast, baseline methods experienced a more significant drop in performance. Mythril, while still the best among baselines, only achieved 70.00% recall and 74.47% precision, indicating a struggle to adapt to unseen patterns. Slither's performance was even lower. GPT-Reentry continued to perform poorly, confirming its limitations in generalization and precision when faced with new contracts. The consistent high performance of **ReDetect** across both the **Ground-truth Dataset** and the **Generalization Dataset** underscores the robustness and adaptability of its hybrid approach, particularly its ability to leverage LLM-derived rules to identify a broader range of reentrancy patterns beyond those explicitly seen during GNN training.

4.7 Runtime Performance

The practical utility of a vulnerability detection tool heavily depends on its efficiency. We evaluated the average analysis time per smart contract for **ReDetect** and the baseline methods on a subset of 100 contracts from our datasets, each with an average of 500 lines of Solidity code. The experiments were conducted on a machine with an Intel Xeon E3-1505M v5 CPU and 32GB RAM. Figure 2 presents the average analysis time.

Figure 2 illustrates that **ReDetect** offers a competitive runtime performance. While slightly slower than purely static analysis tools like Slither (15.2 s), it is significantly faster than symbolic execution tools such as Oyente (68.5 s) and Mythril (55.7 s). The pure LLM approach, GPT-Reentry, exhibited the slowest performance (120.3 s), primarily due to the overhead of complex prompting, token generation, and the iterative nature required for detailed code analysis when relying solely on an LLM.

ReDetect's efficiency stems from the effective division of labor among its components. The GNN efficiently processes the graph representations to identify candidate patterns, which is typically faster than exhaustive symbolic execution. The subsequent static and taint analysis component is designed to be lightweight and focused, only verifying the candidates passed by the GNN, rather than analyzing the entire contract from scratch. This targeted approach allows **ReDetect** to maintain high accuracy without

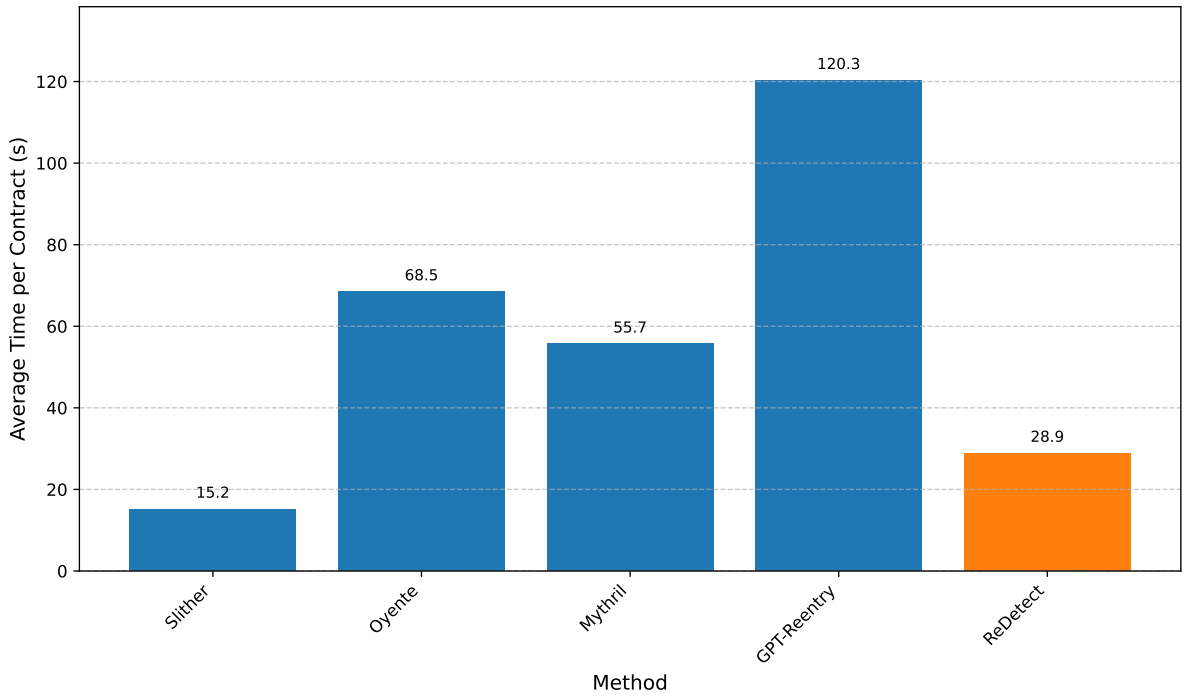

Figure 2: Average Runtime Performance per Smart Contract (100 Contracts, Avg. 500 LoC)

incurring the prohibitive computational costs often associated with more exhaustive analysis techniques. The LLM component's rule generation is performed offline or in a zero/few-shot manner, minimizing online inference latency. This balance of speed and accuracy makes **ReDetect** a highly practical tool for large-scale smart contract auditing.

5 Conclusion

Reentrancy vulnerabilities continue to pose a severe and persistent threat to the security and financial integrity of smart contracts within the rapidly expanding blockchain ecosystem. Despite ongoing research and the development of various detection tools, the inherent complexity of smart contract logic, coupled with the evolving nature of reentrancy attack patterns, has meant that existing static analysis, symbolic execution, and even pure LLM-based approaches often struggle to achieve a desirable balance between high recall and low false positives. This limitation underscores the urgent need for more sophisticated and robust automated detection mechanisms. In response to this critical challenge, we proposed **ReDetect**, a novel and highly effective hybrid framework designed for the efficient and accurate detection of reentrancy vulnerabilities in Ethereum smart contracts. **ReDetect** fundamentally redefines the approach to smart contract security analysis by synergistically integrating the semantic understanding capabilities of Large Language Models (LLMs), the structural pattern recognition power of Graph Neural Networks (GNNs), and the precision of traditional static and taint analysis techniques.

References

[1] Zian Liu, Lei Pan, Chao Chen, Ejaz Ahmed, Shigang Liu, Jun Zhang, and Dongxi Liu. Vulmatch: Binary-level vulnerability detection through signature. *arXiv preprint arXiv:2308.00288v2*, 2023.

[2] Shuo Yang, Jiachi Chen, Mingyuan Huang, Zibin Zheng, and Yuan Huang. Uncover the premeditated attacks: Detecting exploitable reentrancy vulnerabilities by identifying attacker contracts. *arXiv preprint arXiv:2403.19112v1*, 2024.

[3] Kaiwen Wei, Jiang Zhong, Hongzhi Zhang, Fuzheng Zhang, Di Zhang, Li Jin, Yue Yu, and Jingyuan Zhang. Chain-of-specificity: Enhancing task-specific constraint adherence in large language mod-

els. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 2401–2416, 2025.

[4] Gerardo Iuliano, Luigi Allocca, Matteo Cicalese, and Dario Di Nucci. Automated vulnerability injection in solidity smart contracts: A mutation-based approach for benchmark development. *arXiv preprint arXiv:2504.15948v1*, 2025.

[5] Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in ethereum smart contracts. *arXiv preprint arXiv:2105.02881v1*, 2021.

[6] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. Efficiently detecting reentrancy vulnerabilities in complex smart contracts. *arXiv preprint arXiv:2403.11254v1*, 2024.

[7] Pengcheng, Peng, Yun, Qingzhao, Tao, Dawn, Prateek, Sanjeev, Zhuotao, and Xusheng. Contractfix: A framework for automatically fixing vulnerabilities in smart contracts. *arXiv preprint arXiv:2307.08912v2*, 2023.

[8] Peng Qian, Zhenguang Liu, Qinming He, Butian Huang, Duanzheng Tian, and Xun Wang. Smart contract vulnerability detection technique: A survey. *arXiv preprint arXiv:2209.05872v1*, 2022.

[9] Peng Qian, Rui Cao, Zhenguang Liu, Wenqing Li, Ming Li, Lun Zhang, Yufeng Xu, Jianhai Chen, and Qinming He. Empirical review of smart contract and defi security: Vulnerability detection and automated repair. *arXiv preprint arXiv:2309.02391v2*, 2023.

[10] Mohammed Kharma, Soohyeon Choi, Mohammed AlKhanafseh, and David Mohaisen. Security and quality in llm-generated code: A multi-language, multi-model analysis. *arXiv preprint arXiv:2502.01853v1*, 2025.

[11] Qingyuan Li, Binchang Li, Cuiyun Gao, Shuzheng Gao, and Zongjie Li. Empirical study of code large language models for binary security patch detection. *arXiv preprint arXiv:2509.06052v1*, 2025.

[12] Junqiao Wang, Zeng Zhang, Yangfan He, Yuyang Song, Tianyu Shi, Yuchen Li, Hengyuan Xu, Kunyu Wu, Guangwu Qian, Qiuwu Chen, et al. Enhancing code llms with reinforcement learning in code generation. *arXiv preprint arXiv:2412.20367*, 2024.

[13] Kaiwen Wei, Xian Sun, Zequn Zhang, Jingyuan Zhang, Guo Zhi, and Li Jin. Trigger is not sufficient: Exploiting frame-aware knowledge for implicit event argument extraction. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4672–4682, 2021.

[14] Kaiwen Wei, Yiran Yang, Li Jin, Xian Sun, Zequn Zhang, Jingyuan Zhang, Xiao Li, Linhao Zhang, Jintao Liu, and Guo Zhi. Guide the many-to-one assignment: Open information extraction via iou-aware optimal transport. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4971–4984, 2023.

[15] Qiang Yi, Yangfan He, Jianhui Wang, Xinyuan Song, Shiyao Qian, Xinhang Yuan, Miao Zhang, Li Sun, Keqin Li, Kuan Lu, et al. Score: Story coherence and retrieval enhancement for ai narratives. *arXiv preprint arXiv:2503.23512*, 2025.

[16] Haoyu He, Yuede Ji, and H. Howie Huang. Illuminati: Towards explaining graph neural networks for cybersecurity analysis. *arXiv preprint arXiv:2303.14836v1*, 2023.

[17] Varun Gadey, Christoph Sendner Raphael Goetz, Sampo Sovio, and Alexandra Dmitrienko. Gnn-based code annotation logic for establishing security boundaries in c code. *arXiv preprint arXiv:2411.11567v2*, 2024.

[18] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491v3*, 2021.

[19] Noah Ziems and Shaoen Wu. Security vulnerability detection using deep learning natural language processing. *arXiv preprint arXiv:2105.02388v1*, 2021.

[20] Maryam Taeb. Developing hands-on labs for source code vulnerability detection with ai. *arXiv preprint arXiv:2302.00750v1*, 2023.

[21] Gabriele Digregorio, Marco Di Gennaro, Stefano Zanero, Stefano Longari, and Michele Carminati. When secure isn't: Assessing the security of machine learning model sharing. *arXiv preprint arXiv:2509.06703v1*, 2025.

[22] Yucheng Zhou, Lingran Song, and Jianbing Shen. Improving medical large vision-language models with abnormal-aware feedback. *arXiv preprint arXiv:2501.01377*, 2025.

[23] Zhuofu Deng, Yi Xin, Xiaolin Qiu, and Yeda Chen. Weakly and semi-supervised deep level set network for automated skin lesion segmentation. In *Innovation in Medicine and Healthcare: Proceedings of 8th KES-InMed 2020*, pages 145–155. Springer, 2020.

[24] Yi Xin, Juncheng Yan, Qi Qin, Zhen Li, Dongyang Liu, Shicheng Li, Victor Shea-Jay Huang, Yupeng Zhou, Renrui Zhang, Le Zhuo, et al. Lumina-mgpt 2.0: Stand-alone autoregressive image modeling. *arXiv preprint arXiv:2507.17801*, 2025.

[25] Yi Xin, Le Zhuo, Qi Qin, Siqi Luo, Yuewen Cao, Bin Fu, Yangfan He, Hongsheng Li, Guangtao Zhai, Xiaohong Liu, et al. Resurrect mask autoregressive modeling for efficient and scalable image generation. *arXiv preprint arXiv:2507.13032*, 2025.

[26] Haoze Wu, Jiawei Liu, Zheng-Jun Zha, Zhenzhong Chen, and Xiaoyan Sun. Mutually reinforced spatio-temporal convolutional tube for human action recognition. In *IJCAI*, pages 968–974, 2019.

[27] Haoze Wu, Jiawei Liu, Xierong Zhu, Meng Wang, and Zheng-Jun Zha. Multi-scale spatial-temporal integration convolutional tube for human action recognition. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 753–759, 2021.

[28] Junyu Hao, Jianheng Liu, Yongjia Zhao, Zuofan Chen, Qi Sun, Jinlong Chen, Jianguo Wei, and Minghao Yang. Detect an object at once without fine-tuning. In *International Conference on Neural Information Processing*, pages 61–75. Springer, 2024.

[29] Zhihao Lin, Qi Zhang, Zhen Tian, Peizhuo Yu, and Jianglin Lan. Dpl-slam: enhancing dynamic point-line slam through dense semantic methods. *IEEE Sensors Journal*, 24(9):14596–14607, 2024.

[30] Zhihao Lin, Zhen Tian, Qi Zhang, Hanyang Zhuang, and Jianglin Lan. Enhanced visual slam for collision-free driving with lightweight autonomous cars. *Sensors*, 24(19):6258, 2024.

[31] Qinghao Li, Zhen Tian, Xiaodan Wang, Jinming Yang, and Zhihao Lin. Efficient and safe planner for automated driving on ramps considering unsatisfication. *arXiv preprint arXiv:2504.15320*, 2025.

[32] 张左敏and 孔庆峰. 融资约束对企业出口决策的影响——基于银行信用风险的视角. 山东大学学报: 哲学社会科学版, (1):97–105, 2017.

[33] 张左敏. 融资约束对企业出口的影响, 2017.

[34] 张左敏and 孔庆峰. 融资约束对企业出口影响的heckman 验证-基于银行信用风险的视角. 财经理论与实践, (2):23–29, 2017.