Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.074959

ARTICLE

Interpretable Smart Contract Vulnerability Detection with LLM-Augmented Hilbert-Schmidt Information Bottleneck

Yiming Yu[1], Yunfei Guo[2], Junchen Liu[3], Yiping Sun[4] and Junliang Du[5,*]

[1]School of Professional Studies, New York University, New York, NY 10003, USA
[2]Faculty of Computer Science, Dalhousie University, Halifax, NS B3H 4R2, Canada
[3]Department of Computer Science, Boston University, Boston, MA 02215, USA
[4]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, 200240, China
[5]MoE Key Lab of Artifcial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, 200240, China

*Corresponding Author: Junliang Du. Email: jldu@acm.org

Received: 22 October 2025; Accepted: 11 December 2025; Published: 12 March 2026

ABSTRACT: Graph neural networks (GNNs) have shown notable success in identifying security vulnerabilities within Ethereum smart contracts by capturing structural relationships encoded in control- and data-flow graphs. Despite their effectiveness, most GNN-based vulnerability detectors operate as black boxes, making their decisions difficult to interpret and thus less suitable for critical security auditing. The information bottleneck (IB) principle provides a theoretical framework for isolating task-relevant graph components. However, existing IB-based implementations often encounter unstable optimization and limited understanding of code semantics. To address these issues, we introduce ContractGIB, an interpretable graph information bottleneck framework for function-level vulnerability analysis. ContractGIB introduces three main advances. First, ContractGIB introduces an Hilbert–Schmidt Independence Criterion (HSIC) based estimator that provides stable dependence measurement. Second, it incorporates a CodeBERT semantic module to improve node representations. Third, it initializes all nodes with pretrained CodeBERT embeddings, removing the need for hand-crafted features. For each contract function, ContractGIB identifies the most informative nodes forming an instance-specific explanatory subgraph that supports the model's prediction. Comprehensive experiments on public smart contract datasets, including ESC and VSC, demonstrate that ContractGIB achieves superior performance compared to competitive GNN baselines, while offering clearer, instance-level interpretability.

KEYWORDS: Smart contract vulnerability detection; graph neural networks; information bottleneck; Hilbert-Schmidt Independence Criterion (HSIC)

1 Introduction

The maturation of blockchain ecosystems has accelerated the adoption of smart contracts self-executing programs that encode and enforce agreements without intermediaries. As these contracts increasingly underpin decentralized finance (DeFi), digital asset management, supply-chain automation, and a wide range of Web3 applications, their correctness and robustness have become essential to the security of the underlying platforms [1]. However, the immutable and autonomous nature of smart contracts implies that design flaws, logic errors, or subtle misuse of platform features can lead to substantial financial losses, breaches of confidentiality, and cascading failures across decentralized systems [2]. The steady rise in high-impact security incidents, many of which exploit nuanced interactions between control-flow, state updates, and external calls, underscores the pressing need for principled and effective methods to identify vulnerabilities before contracts are deployed on-chain.

A large body of prior work on smart contract security has relied on program analysis techniques to identify potential vulnerabilities. Classical tools such as Mythril [3], Oyente [4], SmartCheck, and Securify employ static analysis, symbolic execution, formal verification, or fuzzing to search for violations of predefined security properties. These methods are particularly effective for detecting well-understood issues, including reentrancy, integer overflows, and unauthorized access, by matching contract behaviors against handcrafted rules or symbolic constraints. Although efficient for common vulnerability patterns, such approaches often struggle to generalize to increasingly complex Solidity constructs and frequently exhibit limited scalability. Large-scale empirical studies report that their recall rates typically remain below 60%, reflecting the difficulty of achieving comprehensive coverage in real-world deployments.

Motivated by these limitations, recent research has turned toward data-driven methods that learn vulnerability patterns from labeled examples. Machine learning and deep learning approaches [5–8] have demonstrated promising results by extracting statistical regularities directly from source code. However, many existing models treat the contract as a linear sequence of tokens or statements, which neglects the rich structural information that governs execution behavior. Such flattened representations fail to account for critical control-flow and data-flow interactions that often determine whether a contract is exploitable.

To more faithfully capture these dependencies, smart contracts can be modeled as graphs in which nodes represent program entities (e.g., functions, operations, or state variables) and edges encode semantic relations such as control transfer or data dependencies. This representation naturally enables the use of graph neural networks (GNNs) [9], which propagate contextual information through graph neighborhoods and have shown strong performance in smart contract analysis tasks [10,11]. By learning over structured program graphs, GNNs can jointly model both local execution context and global logic across a contract [12–14].

Despite these advantages, GNN-based detectors remain difficult for auditors to trust in practice. Their predictions are typically opaque, offering little insight into which parts of the contract contribute to a detected vulnerability. This lack of interpretability has been widely recognized as a barrier to adoption in auditing pipelines [15]. In security workflows, analysts require fine-grained explanations, often at the subgraph or statement level, to validate model outputs and prioritize remediation efforts. Improving the interpretability of GNN predictions therefore plays a crucial role in bridging the gap between automated analysis and practical auditing needs.

The information bottleneck (IB) principle [16] provides a powerful formalism for learning compact yet task-relevant representations. In the context of smart contract analysis, the IB objective can be viewed as identifying a minimal substructure, such as a subgraph, that preserves the information most predictive of a potential vulnerability. This perspective offers a principled alternative to *post-hoc* explanation techniques by embedding interpretability directly into the model design: the learned representation itself must discard irrelevant details while retaining the features that truly influence the final prediction. Recent developments have adapted the IB framework to graph domains, such as the Subgraph Information Bottleneck (SIB) [17], which seeks minimal predictive subgraphs in molecular tasks.

A central challenge in deploying graph-based IB methods lies in mutual information estimation. Existing approaches typically rely on variational bounds optimized through auxiliary neural networks [17,18], or compute matrix-based estimators that require eigenvalue decomposition and differentiation over spectral components [19]. Both strategies often lead to training instability, sensitivity to hyperparameters, and increased computational overhead.

The emergence of large language models (LLMs) introduces new opportunities in this setting. When contracts are encoded using pretrained models such as CodeBERT [20], the node features already contain rich semantic priors extracted from large-scale code corpora. This semantic structure naturally complements the IB objective: with more informative node embeddings, the model can focus on identifying which *regions* of the contract carry causal signals for the vulnerability, rather than expending capacity on learning basic syntactic or lexical patterns. However, these high-level embeddings also amplify the need for stable and tractable mutual information estimators, as noisy or unbalanced gradients can easily disrupt the interaction between semantic and structural representations. This motivates the development of simplified, more reliable IB objectives tailored for graph models operating on LLM-enhanced features.

This paper addresses the above limitations through the following key contributions:

- We propose ContractGIB, a novel framework that applies the information bottleneck (IB) principle to enable interpretable GNN-based detection of smart contract vulnerabilities. To the best of our knowledge, this is the first framework that integrates the IB principle into graph neural networks for this task. In addition, our design incorporates large language model (LLM) derived semantic representations into the graph construction pipeline, enabling richer contract semantics to be captured within the IB-based interpretability mechanism.
- A dual-branch GNN encoder and an HSIC-based information bottleneck: We design a hybrid encoder that jointly models structural dependencies and LLM-derived semantic cues through a dual-branch architecture, enabling richer reasoning over both control/data-flow and semantic interactions. To improve stability and interpretability, we formulate an HSIC-driven information bottleneck objective that avoids auxiliary discriminators and spectral decompositions required by mutual-information estimators, leading to more reliable optimization.
- Comprehensive empirical validation and enhanced interpretability: Experiments on two benchmark datasets and 12 competing methods show that LLM-guided semantics substantially improve both predictive performance and explanation quality. The proposed framework yields more stable training behavior, more expressive subgraph-based explanations, and consistently higher detection accuracy across vulnerability categories.

2 Related Work

This section surveys prior studies on smart contract vulnerability detection, covering both classical program analysis techniques and modern neural learning approaches in Section 2.1. In Section 2.2, we further outline recent progress on incorporating IB principles to improve model effectiveness and interpretability.

2.1 Existing Approaches in Smart Contract Vulnerability Detection

Early research on smart contract security has been dominated by analysis techniques that reason directly over program execution behaviors, which can be broadly divided into static and dynamic categories. Static approaches aim to infer vulnerabilities without executing the contract, typically through symbolic reasoning, semantic program analysis, or formal verification. For example, Luu et al. [4] employed symbolic execution to systematically explore execution paths and identify violations of security rules specified by domain experts. Tsankov et al. [21] developed a semantic analysis framework that derives vulnerability-related facts by reasoning over data-flow dependencies in contract programs. In contrast, dynamic analysis techniques detect vulnerable behaviors by executing the contract and observing its runtime interactions. Nguyen et al. [22] proposed Serum, which utilizes dynamic taint tracking to trace information propagation during execution and uncover sophisticated attack patterns. Smartian [23] further enhances this paradigm by guiding fuzz testing with data-flow feedback, enabling the discovery of transaction sequences that activate deeper contract states and expose previously hidden vulnerabilities.

Beyond these classical tools, recent research has produced several scalable pipelines for analyzing EVM bytecode. Representative examples include EtherSolve [24], EVMLiSA [25], GigaHorse [26], EthIR [27], and eThor [28]. These systems lift EVM bytecode into intermediate representations that support data-flow and control-flow reconstruction, enabling analysis even when Solidity source code is unavailable. On the other hand, Albert et al. [29] provides a complementary perspective by formalizing callback safety and identifying modularity violations, a topic strongly related to reentrancy and cross-function interaction risks.

Learning-based techniques for smart contract vulnerability detection have been investigated from multiple perspectives, spanning token-sequence modeling and program-representation learning. Early studies predominantly treated contract code as natural language–like sequences and applied neural language models to extract semantic features. For example, Gogineni et al. [30] adapted the AWD-LSTM architecture by replacing its decoder with a discriminative classification head, enabling the model to learn vulnerability-aware representations directly from source code. In a complementary line of work, SoliAudit [31] integrated static analysis with runtime fuzzing, employing Word2Vec embeddings for code representation [32] and a logistic regression classifier for vulnerability prediction, thereby improving robustness against both known and previously unseen attack patterns. Despite these advances, sequence-based learning frameworks largely ignore the explicit program structures that govern execution behavior. In particular, critical control-flow and data-flow dependencies across functions and statements are not explicitly modeled, which limits the ability of such approaches to faithfully capture the semantic conditions under which many vulnerabilities arise.

To overcome the limitations of sequence-based models, recent work has increasingly adopted graph representations for smart contract analysis. Eth2Vec [33] leverages abstract syntax trees to model syntactic structure, while more advanced approaches explicitly encode control-flow and data-flow dependencies into program graphs, enabling GNN-based learning for vulnerability detection [10,11,34]. By exploiting execution-level structure, these methods achieve notably stronger detection performance than token-based models. Nevertheless, most existing GNN-based detectors remain opaque: they provide little explanation of which nodes, edges, or substructures drive the final prediction. This lack of interpretability severely limits their reliability and hinders deployment in practical security auditing workflows. Given the diversity of existing approaches, it is useful to contrast the characteristics of source code analysis and bytecode analysis. Table 1 presents a concise comparison.

Table 1: Comparison of solidity source code analysis and EVM bytecode analysis

Aspect	Solidity (Source code)	EVM bytecode
Availability	Often unavailable for deployed contracts	Always available on-chain
Semantic information	High-level constructs and types preserved	Semantics lost after compilation
Analysis complexity	Easier to analyze due to structured control flow	Harder due to unstructured jumps and indirect calls
Robustness	Sensitive to source-level obfuscation and inline assembly	Independent of source-level obfuscation

Recent studies also explore large language models (LLMs) for vulnerability detection, combining semantic reasoning and code understanding [35,36]. Although promising, LLM-based approaches still face challenges in handling control-flow structure and producing faithful explanations.

Our framework operates primarily on structured graphs derived from source code while enriching node representations with CodeBERT semantics. This design leverages the higher level reasoning afforded by source code, while avoiding the brittleness of rule based detectors and the opacity of standard GNNs.

The integration of information bottleneck ideas enables instance specific subgraph explanations that enhance interpretability.

2.2 Information Bottleneck and Its Applications in Graph Data

The IB principle [16] provides a theoretical framework for learning representations that retain only the information in a random variable X that is most relevant for predicting a target variable Y. This is achieved by introducing a compressed latent representation $\tilde{X}$ that maximizes the predictive mutual information $I(\tilde{X};Y)$ while simultaneously restricting the retained information about the input through $I(\tilde{X};X)$:

$$\mathcal{L}[p(\tilde{x}|x)] = \min I(\tilde{X};X) - \beta I(\tilde{X};Y), \tag{1}$$

where $\beta \geq 0$ is a Lagrange multiplier. By minimizing the $\mathcal{L}[p(\tilde{x}|x)]$, the IB principle also provides a natural approximation of minimal sufficient statistic [37].

The IB principle has recently been extended to graph-structured learning problems, with applications spanning both graph representation learning and model interpretability [17,38]. A representative example is the Subgraph Information Bottleneck (SIB) framework proposed by Yu et al. [17], which formulates graph explanation as the problem of selecting a minimal yet label-informative subgraph. Rather than operating on the full input graph, SIB explicitly learns a subgraph G_{sub} that balances compactness and predictive relevance. Its objective is defined as

$$\mathcal{L}_{\text{SIB}} = \min I(G;G_{\text{sub}}) - \beta I(Y;G_{\text{sub}}), \tag{2}$$

where the first term enforces compression of the original graph and the second term encourages the preservation of label-relevant information. By directly optimizing for an explanatory subgraph during training, SIB yields a graph neural network with intrinsic interpretability rather than relying on *post-hoc* explanation methods.

In SIB [17] and related follow-up studies [39,40], the dependency between the original graph G and the selected subgraph G_{sub} is typically quantified through the Mutual Information Neural Estimator (MINE) [18]. MINE introduces an auxiliary critic network to maximize a variational lower bound of mutual information based on the Donsker-Varadhan representation. Although flexible, this adversarial optimization strategy is known to suffer from high gradient variance and sensitivity to hyperparameters, which often results in unstable training behavior in practice.

A different family of estimators is built upon the matrix-based Rényi α-entropy functional [19], which provides a non-parametric alternative without relying on adversarial learning. However, this formulation requires performing eigenvalue decomposition on an $N \times N$ kernel matrix, leading to a cubic computational complexity of $\mathcal{O}(N^3)$ with respect to the sample size. As a result, its computational overhead becomes prohibitive for large-scale graph datasets commonly encountered in real-world applications.

3 Our Method

3.1 Problem Formulation

Given the source code of a smart contract, our objective is to perform vulnerability detection at the granularity of individual functions. Each function instance is formulated as a binary classification task, where the predicted label $\hat{y} \in \{0,1\}$ indicates whether the function exhibits a specific vulnerability type ($\hat{y} = 1$) or is considered secure ($\hat{y} = 0$).

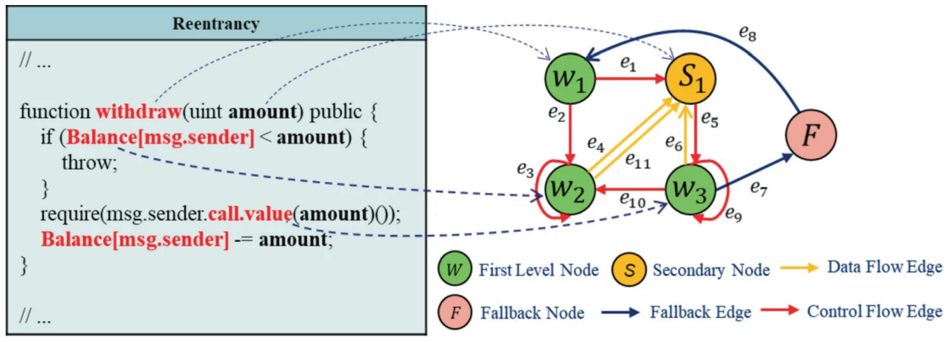
To capture execution-relevant structure, each function is first transformed into a contract graph as described in Section 3.2. Let $\{G^1, G^2, \ldots, G^N\}$ denote the set of resulting graphs for N function instances.

Our proposed framework, ContractGIB, maps each G^i to a corresponding prediction $\hat{y}^i$. Beyond label prediction, the model also produces an explanation for each decision by identifying a compact, task-relevant subgraph G^i_{sub} that highlights the nodes most responsible for the detected vulnerability.

This study concentrates on two representative vulnerability categories. *Reentrancy* arises when a contract transfers Ether to an external address before its internal state is securely updated, allowing the callee to re-invoke the original function and repeatedly manipulate the same execution context. Such reentrant behaviors can enable an attacker to drain funds through recursive external calls before the initial invocation completes. *Infinite-loop vulnerabilities* stem from execution paths that fail to reach a termination condition, resulting in unbounded computation. In Ethereum, this problem is further amplified by the interaction between regular functions and fallback functions. Cyclic call patterns can emerge when a function triggers a fallback routine, which then redirects control back to the original function, forming a non-terminating call sequence that exhausts gas and disrupts contract behavior.

3.2 Graph Construction

A contract function contains heterogeneous program elements that contribute unequally to vulnerability formation. Following the high-level structure of prior work [10,41] while introducing a stronger semantic foundation, we construct a semantic-structural contract graph $G = (X, A)$ that preserves the roles of three node categories: *major nodes*, *secondary nodes*, and *fallback nodes*, as illustrated in Fig. 1a.

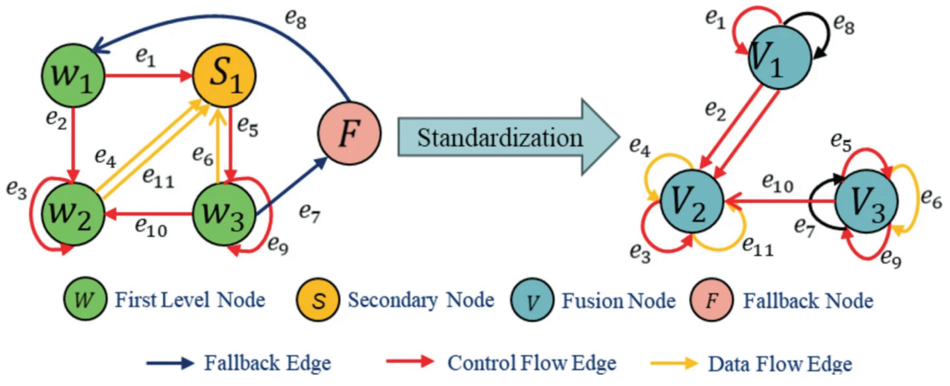


```
// ...

function withdraw(uint amount) public {
    if (Balance[msg.sender] < amount) {
        throw;
    }
    require(msg.sender.call.value(amount)());
    Balance[msg.sender] -= amount;
}

// ...
```

(a) Example Solidity function and its corresponding initial contract graph, where nodes represent major operations (W), secondary variables/intermediate values (S), and the fallback behavior (F), and edges encode control-flow, data-flow, and fallback/callback relations.



(b) Standardization (normalization) of the contract graph by eliminating secondary/fallback nodes and propagating their information to adjacent major nodes, yielding a compact fused-node graph (V) used as the input to ContractGIB.

Figure 1: Smart contract graph generation and normalization

Major nodes correspond to operations whose behavior is strongly tied to vulnerability types, including external calls, financial transfers, and state-updating instructions. These nodes often represent the key steps in vulnerability-triggering execution chains. Secondary nodes capture storage variables and intermediate values (e.g., balance fields, lock flags, control signals) that influence or condition the behavior of major nodes. In addition, Ethereum contracts may implicitly trigger fallback logic during execution; to model this behavior, we introduce a dedicated fallback node that represents attacker-side fallback activation, which plays a crucial role in reentrancy and callback-based exploits.

Edges are added between nodes to reflect possible execution and dependency relationships. We include control-flow edges, data-flow edges, invocation edges, and implicit semantic edges that account for reentrant callbacks, modifier-induced control transfers, and other behaviors detectable through semantic similarity. Each edge is represented as (v_s, v_e, t, o), where t denotes the dependency type and o indicates execution order.

To obtain a compact and interpretable graph for downstream processing, we perform node elimination with semantic aggregation. In this step, secondary nodes and the fallback node are removed from the graph, and their semantic and structural information is propagated to adjacent major nodes through an attention-based aggregation rule:

$$x_i' = x_i + \sum_{j \in \mathcal{N}(i)} \alpha_{ij} x_j, \qquad \alpha_{ij} = \frac{\exp(\phi(x_i, x_j))}{\sum_{k \in \mathcal{N}(i)} \exp(\phi(x_i, x_k))}. \tag{3}$$

Here, $\mathcal{N}(i)$ denotes the set of eliminated secondary or fallback nodes adjacent to the major node v_i. The compatibility function $\phi(x_i, x_j)$ is defined as an additive attention scoring function:

$$\phi(x_i, x_j) = \mathbf{a}^\top \tanh\big(W\,[x_i \parallel x_j]\big), \tag{4}$$

where W and $\mathbf{a}$ are trainable parameters and $[x_i \parallel x_j]$ denotes the concatenation of the feature vectors of nodes i and j. This formulation assigns higher attention weights to auxiliary nodes whose semantic and structural features are more relevant to the major node, enabling effective information propagation during node elimination.

This aggregation step preserves the contextual influence of supporting nodes while merging their information into the major nodes, producing the normalized structure depicted in Fig. 1b. The resulting graph retains essential vulnerability signals—both structural and semantic—while avoiding redundancy and fragmentation, and serves as the input to the dual-branch encoder described later.

3.2.1 LLM-Assisted Semantic Node Annotation

While the above construction provides a structural foundation, it relies on manually defined heuristics and may overlook subtle semantic cues. We therefore introduce an LLM-assisted semantic annotation module, which leverages CodeBERT [20], a large language model pretrained on source code including Solidity-like smart contracts. For each code fragment corresponding to a node, we compute an embedding

$$e_i = f_{\text{CodeBERT}}(c_i), \tag{5}$$

that captures both syntactic roles and vulnerability-related semantics. The final node feature becomes:

$$\tilde{x}_i = [x_i \oplus e_i], \tag{6}$$

where x_i is the original structural feature and $\oplus$ denotes concatenation. For instance, two nodes may both contain `call.value`, but CodeBERT embeddings can distinguish between guarded transfers (safe) and unguarded transfers (risky), providing richer vulnerability-sensitive features.

3.2.2 LLM-Based Node Embedding Initialization

In addition to semantic annotation, we initialize node embeddings directly from CodeBERT rather than relying on handcrafted or one-hot encodings. For each node v_i with associated code snippet c_i, we compute:

$$x_i^{(0)} = f_{\text{CodeBERT}}(c_i). \tag{7}$$

This initialization ensures that nodes start with semantically meaningful representations aligned with CodeBERT's understanding of code. Unlike purely structural features, CodeBERT embeddings encode high-level programming semantics and security-related cues. We allow these embeddings to be fine-tuned during GNN training, so that they adapt to the vulnerability detection task. Empirically, this initialization accelerates convergence and improves robustness against code obfuscation.

3.2.3 Integration into ContractGIB

Together, these two CodeBERT-enhanced modules, i.e., semantic node annotation and embedding initialization, complement our graph construction. They enable ContractGIB to incorporate domain knowledge from pretrained language models into node representation and structural reasoning, thereby improving detection accuracy and interpretability.

3.3 Framework of ContractGIB

ContractGIB consists of four key modules: a subgraph generator, a graph encoder, a mutual information estimator, and a predictor (see Fig. 2). The Subgraph Generator samples subgraphs G_{sub} from the original graph G. The graph encoder extracts graph embeddings from either G or G_{sub}. The mutual information estimator assesses the mutual information between G and G_{sub}.

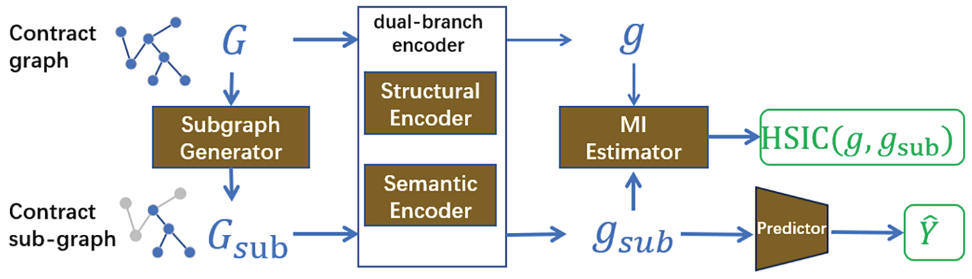


Figure 2: Framework of our proposed ContractGIB, which consists of four modules, including a subgraph generator, a graph encoder, and a mutual information estimator, and a predictor. Given the input contract graph G, the subgraph generator samples a contract subgraph G_{sub} along with the node assignment, indicating whether a specific node belongs to G_{sub} or $\overline{G_{\text{sub}}}$. The dual-branch semantic-structural encoder is used to learn graph embeddings g and g_{sub} from G and G_{sub}, respectively. The mutual information estimator assesses the mutual information between $I(G_{\text{sub}}, G)$ between G and G_{sub} with the mutual information neural estimator (MINE). Finally, g_{sub} is used to predict $\hat{Y}$

3.3.1 Subgraph Generator

Given an input contract graph $G = (X, A)$, the goal of the subgraph generator is to identify a predictive subgraph G_{sub} by learning a node-selection distribution. Each node V_i is associated with a two-dimensional vector $S_i \in \mathbb{R}^2$, where the entries correspond to the probabilities of assigning the node to the retained subgraph G_{sub} or to its complement $\overline{G_{\text{sub}}}$, respectively. Collecting these row-wise assignments yields a matrix $S \in \mathbb{R}^{n\times 2}$ for a graph with n nodes, following the probabilistic formulation used in [17].

To compute S, we first propagate node features, combining both structural descriptors and LLM-derived semantic embeddings, through an l-layer graph encoder. Let $H^{(0)} = X$ denote the initial node representations. The encoder produces intermediate embeddings through:

$$H^{(l)} = \text{GCN}(A, H^{(l-1)}; \theta_1), \tag{8}$$

where θ_1 denotes the trainable parameters of the structural branch.

The final embeddings $H^{(l)}$ are then transformed by a lightweight multilayer perceptron to obtain node-level selection logits:

$$S = \text{Softmax}\big(\text{MLP}(H^{(l)}; \theta_2)\big), \tag{9}$$

with θ_2 representing the parameters of the MLP.

A row-wise softmax ensures that each node receives a normalized probability distribution over the two possible assignments. During training, the IB objective encourages S to evolve toward near-deterministic values (i.e., assignments close to 0 or 1), yielding a sparse and interpretable mask that highlights the subgraph most responsible for the model's prediction. This probabilistic formulation enables end-to-end optimization while providing a principled mechanism for learning task-relevant subgraph structures.

The pseduo code for subgraph generation is summarized in Algorithm 1.

Algorithm 1: Node assignment and subgraph generation

```
Require: Contract graph $G = (V, E)$ with $n$ nodes, adjacency matrix $A$, initial node features $X^0$, GCN parameters $\theta_1$, MLP parameters $\theta_2$, number of GCN layers $L$
Ensure: Node assignment matrix $S \in \mathbb{R}^{n\times 2}$ and subgraph $G_{\text{sub}}$
 1: $X \leftarrow X^0$
                                                    ▷ GCN encoder
 2: for $\ell = 1$ to $L$ do
 3:     $X \leftarrow \text{GCNLayer}(A, X; \theta_1^{(\ell)})$
 4: end for
 5: $X^L \leftarrow X$
                ▷ MLP + row-wise softmax to obtain assignment probabilities
 6: $Z \leftarrow \text{MLP}(X^L; \theta_2)$                      ▷ $Z \in \mathbb{R}^{n\times 2}$
 7: $S \leftarrow \text{RowSoftmax}(Z)$   ▷ $S_{i,:} = [p(V_i \in G_{\text{sub}}), p(V_i \in \overline{G_{\text{sub}}})]$
                           ▷ Optional hard assignment (for extraction)
 8: for each node $V_i \in V$ do
 9:     $\text{label}_i \leftarrow \arg\max_{c\in\{1,2\}} S_{i,c}$      ▷ 1: $G_{\text{sub}}$, 2: $\overline{G_{\text{sub}}}$
10: end for
11: $V_{\text{sub}} \leftarrow \{V_i \in V \mid \text{label}_i = 1\}$
12: $G_{\text{sub}} \leftarrow G[V_{\text{sub}}]$                ▷ Induced subgraph
13: return $S, G_{\text{sub}}$
```

3.3.2 Dual-Branch Semantic-Structural Graph Encoder

The goal of the graph encoder is to derive a compact graph-level representation from the contract graph $G = (\tilde{X}, A)$, where $\tilde{X}$ denotes the semantic-structural node features and A encodes execution-related dependencies. To address the limitations of purely structural GNNs, we develop a dual-branch semantic-structural encoder that jointly reasons about (i) fine-grained control-flow and data-flow relations, and (ii) high-level semantic cues extracted from source code. The encoder is applied to both the full graph G and the generated subgraph G_{sub}, producing graph-level embeddings for downstream prediction and interpretability.

The *structural branch* models local and multi-hop dependencies through an enhanced graph convolutional mechanism:

$$H_{\text{str}}^{(l+1)} = \sigma\left(\hat{A}H_{\text{str}}^{(l)} W^{(l)} + H_{\text{str}}^{(l)} U^{(l)}\right), \tag{10}$$

where $\hat{A} = A + A^2 + I$ incorporates higher-order execution paths, $W^{(l)}$ and $U^{(l)}$ are learnable parameters, and $\sigma(\cdot)$ denotes a nonlinear activation. The residual/global-jump term $H_{\text{str}}^{(l)} U^{(l)}$ mitigates oversmoothing and helps preserve long-range dependency patterns frequently observed in smart contract behavior. We initialize the branch by setting

$$H^{(0)} = \tilde{X}, \tag{11}$$

where $\tilde{X}$ includes both structural descriptors and CodeBERT-derived semantic embeddings.

The *semantic branch* processes the same input $\tilde{X}$ using a lightweight graph Transformer:

$$H_{\text{sem}} = \text{GT}_{\text{light}}(\tilde{X}, A), \tag{12}$$

in which attention weights are computed under the structural constraints imposed by A. This branch captures semantic interactions that are difficult to infer solely from graph structure, including modifier-induced control changes, implicit fallback activation, and semantic relationships across distant code regions.

The outputs from the structural and semantic branches are fused through a learnable gate:

$$H = \gamma \cdot H_{\text{sem}} + (1 - \gamma) \cdot H_{\text{str}}, \qquad \gamma \in [0, 1]. \tag{13}$$

Finally, a second-order bi-linear pooling layer aggregates the fused node representations into a compact graph embedding:

$$g = \text{flatten}(\text{SOPOOL}_{\text{bi-linear}}(H)). \tag{14}$$

The resulting representation is used both for prediction and for the information bottleneck objectives described in the next section.

3.3.3 Dependence Estimator: Hilbert–Schmidt Independence Criterion (HSIC)

To instantiate the compression term in the information bottleneck, we replace mutual information with the *Hilbert–Schmidt Independence Criterion* (HSIC) [42], a kernel-based dependence measure that is differentiable, parameter-free (no discriminator network), and stable for mini-batch training. For characteristic kernels, HSIC is zero if and only if the variables are independent, making it a suitable surrogate for $I(\cdot;\cdot)$.

Let $\{(g_i, g_i^{\text{sub}})\}_{i=1}^{B}$ be a mini-batch of size B, where g_i and g_i^{sub} are graph-level embeddings of the full graph G and the subgraph G_{sub} produced by our dual-branch encoder (with CodeBERT-based initialization and annotation; see Sections 3.2 and 3.3.2). Define positive semi-definite kernels k and ℓ on these embeddings, with Gram matrices $K, L \in \mathbb{R}^{B\times B}$ given by $K_{ij} = k(g_i, g_j)$ and $L_{ij} = \ell(g_i^{\text{sub}}, g_j^{\text{sub}})$. Let $H = I - \frac{1}{B}\mathbf{1}\mathbf{1}^\top$ be the centering matrix.

The commonly used (biased) empirical HSIC is

$$\widehat{\text{HSIC}}_{\text{b}}(g, g_{\text{sub}}) = \frac{1}{(B-1)^2}\,\text{tr}(KHLH). \tag{15}$$

It is differentiable with respect to K and L, and thus with respect to the embeddings. To make the scale invariant and to bound the term into $[0, 1]$, we adopt the *normalized HSIC* (nHSIC):

$$\text{nHSIC}(g, g_{\text{sub}}) = \frac{\text{tr}(KHLH)}{\sqrt{\text{tr}(KHKH)\,\text{tr}(LHLH)} + \varepsilon}, \tag{16}$$

where $\varepsilon > 0$ avoids division by zero. An unbiased estimator [42] may also be used; in practice we found the biased/nHSIC variants more stable under SGD.

In our paper, we use Gaussian (RBF) kernels

$$k(x,x)' = \exp\Big(-\tfrac{\|x-x'\|_2^2}{2\sigma_k^2}\Big), \qquad \ell(z,z)' = \exp\Big(-\tfrac{\|z-z'\|_2^2}{2\sigma_\ell^2}\Big), \tag{17}$$

with bandwidths selected by the median heuristic within each batch. To reduce bandwidth sensitivity, we employ a *multi-kernel* variant (MK-HSIC): $K = \sum_{s\in\mathcal{S}} K^{(s)}$ with $\sigma_k^{(s)} \in \{0.5m, m, 2m\}$, where m is the batch median pairwise distance (similarly for L).

3.3.4 Final GIB Objective with HSIC

Motivated by the HSIC bottleneck [43], we train ContractGIB with the following objective:

$$\mathcal{L}(\theta) = \underbrace{\mathcal{L}_{\text{CE}}(\widehat{Y}, Y)}_{\text{prediction}} + \lambda \underbrace{\text{nHSIC}(g, g_{\text{sub}})}_{\text{compression}} - \beta \underbrace{\text{nHSIC}(g_{\text{sub}}, Y)}_{\text{predictive dependence}}, \tag{18}$$

where $\mathcal{L}_{\text{CE}}$ is the cross-entropy loss. Note that, minimizing the cross-entropy loss $\min \mathcal{L}_{\text{CE}}(\hat{Y}, Y)$ can be considered as an approximation to maximizing the mutual information term $I(G_{\text{sub}}; Y)$ in Eq. (2) [17].

$\lambda > 0$ controls compression strength, and $\beta \geq 0$ optionally encourages stronger dependence between the subgraph representation and the labels. For the label dependence, we use a delta kernel on labels, $\ell_y(y_i, y_j) = \mathbb{I}[y_i = y_j]$ (or, for soft labels, a linear kernel on class-probability vectors). In ablations we report results both with and without the predictive HSIC term (setting $\beta = 0$ reduces Eq. (18) to CE + compression only).

For Eq. (15), the gradients with respect to the Gram matrices are

$$\frac{\partial \widehat{\text{HSIC}}_{\text{b}}}{\partial K} = \frac{1}{(B-1)^2} HLH, \qquad \frac{\partial \widehat{\text{HSIC}}_{\text{b}}}{\partial L} = \frac{1}{(B-1)^2} HKH. \tag{19}$$

For RBF kernels, $\partial K_{ij}/\partial g_i = K_{ij}\,(g_j - g_i)/\sigma_k^2$ (and analogously for L), making the estimator end-to-end differentiable. We compute HSIC per mini-batch; all operations are GPU-friendly matrix multiplications.

HSIC requires $\mathcal{O}(B^2)$ time and memory per batch to form K and L. We use batch sizes $B \in [128, 512]$ without issues on a 24 GB GPU. For larger batches, one can (i) compute HSIC on blocks and average (block-HSIC), or (ii) approximate RBF kernels via random Fourier features or Nyström projections.

Unless otherwise noted, we use: (i) ℓ_2-normalized embeddings g and g_{sub}; (ii) MK-RBF kernels with three bandwidths via the median heuristic; (iii) normalized HSIC (Eq. (16)) with $\varepsilon = 10^{-8}$;

(iv) hyperparameters $\lambda \in \{0.1, 0.3, 1.0\}$ and $\beta \in \{0, 0.1, 0.3\}$ tuned on the validation set. The CE term prevents representational collapse while the compression term enforces sparsity of information flow from G to G_{sub}.

Compared to discriminator-based MI bounds (e.g., MINE) or spectral entropy estimators, HSIC avoids adversarial training and eigen-decompositions while retaining a principled independence target. In our setting it serves as a stable, plug-and-play estimator that integrates seamlessly with the CodeBERT-augmented encoder and the subgraph generator.

4 Experiments

4.1 Datasets

We evaluate ContractGIB on two widely used datasets for graph neural network based smart contract vulnerability detection, namely the Ethereum Smart Contract dataset (ESC) and the VNT Smart Contract dataset (VSC). These two datasets are popular choices in recent GNN oriented vulnerability detection research. For example, the pioneering GNN based work Smart Contract Vulnerability Detection Using Graph Neural Networks [10] adopts both ESC and VSC as benchmark datasets. Following these established datasets ensures a fair comparison with existing GNN based approaches and maintains consistency with prior work in this research line. This choice is further supported by recent studies in the field. Both Dong et al. [44] and Gupta et al. [45] use the ESC and VSC datasets when evaluating their GNN based vulnerability detection models, which confirms that these two datasets have become widely accepted benchmarks for graph oriented approaches. Both datasets are publicly available from https://github.com/vntchain/GNNSCVulDetector/tree/master.

ESC (Ethereum Smart Contracts). The ESC dataset contains 40,932 real-world Solidity smart contracts collected from the Ethereum blockchain. Each contract is labeled for reentrancy based on a combination of automated static analysis and manual verification conducted in the original dataset construction process. We directly reuse the verified labels provided by the dataset creators and do not introduce any new heuristic labeling. Because the dataset consists of real deployed contracts at scale, it provides a diverse and realistic benchmark for evaluating reentrancy detection.

VSC (VNT Smart Contracts). The VSC dataset includes 4170 C-style smart contracts deployed on the VNT chain. It contains function-level annotations for infinite loop vulnerabilities generated using the hybrid static analysis pipeline of the original benchmark, supplemented by manual inspection. Similar to ESC, we directly use these existing ground truth labels without modification. The VSC dataset complements ESC by covering a different contract language and execution environment, thereby enabling cross-platform evaluation of ContractGIB.

Across both datasets, we adopt an 80/20 training-test split following prior work. All experiments strictly use the original labels and structures provided by the benchmark datasets to ensure the validity and reproducibility of our evaluation.

The authors are aware of two additional datasets frequently referenced in the vulnerability-detection community, namely SmartBugs [46] and SolidiFI [47]. These datasets contain high-quality, curated vulnerability examples, but they are primarily designed for benchmarking analysis tools rather than training or evaluating graph neural network models. Because they do not provide function-level graph structures, they are not suitable for our graph-based learning framework.

4.2 Comparison with Existing Methods

We evaluate the proposed ContractGIB framework against a diverse set of competing methods, including twelve representative approaches and a strong graph-based baseline, DR-GCN. The selected baselines cover three major categories. The first group consists of established symbolic and rule-based analyzers for smart contracts, including Oyente [4], Mythril [3], SmartCheck [48], and Securify [21]. The second group includes neural learning models that operate on sequential or graph-structured representations, namely Vanilla-RNN, LSTM, GRU, and GCN. The third group comprises dedicated program loop detection techniques, including Jolt [49], PDA [50], SMT [51], and Looper [52]. Together, these baselines provide a comprehensive and challenging benchmark for evaluating vulnerability detection performance. Following recommendations by [53], we also include Pluto [54] as a modern formal-methods baseline. Pluto provides reentrancy reports at the bytecode level, which allows for a qualitative comparison. Our hyperparameters are summarized in Table 2.

Table 2: Key hyperparameters used in all experiments

Parameter	Value
Learning rate	1e−4
Batch size	32
GIB coefficient β	0.1
HSIC weight λ	0.2
Kernel bandwidths	median heuristic
Subgraph size k	20% of nodes

The evaluation considers four metrics: accuracy, recall, precision, and F1 score. The GPU setup used in the experiments consists of an NVIDIA GeForce RTX 3080Ti, running in an environment configured with PyTorch 2.1.0 and CUDA 12.1 (cu121). The quantitative results reported in Tables 3 and 4 show that *ContractGIB* delivers the strongest overall performance on both vulnerability detection tasks. Across reentrancy and infinite-loop benchmarks, it consistently surpasses all compared approaches, including competitive graph-based models such as DR-GCN and GCN, as well as widely used static analyzers like Securify and Mythril. In particular, *ContractGIB* achieves the best balance between precision and recall, attaining the top F1 scores of 78.40% for reentrancy and 69.12% for infinite-loop detection, which indicates robust and reliable predictive capability.

Table 3: Performance comparison on reentrancy vulnerability detection. The best performance is in bold

Methods	Acc (%)	Recall (%)	Precision (%)	F1 (%)
Smartcheck	52.97	32.08	25.00	28.10
Oyente	61.62	54.71	38.16	44.96
Mythril	60.54	71.69	39.58	51.02
Securify	71.89	56.60	50.85	53.57
Pluto	76.40	78.55	70.10	73.85
Vanilla-RNN	49.64	58.78	49.82	50.71
LSTM	53.68	67.82	51.65	58.64
GRU	54.54	71.30	33.10	45.60
GCN	77.85	78.79	70.02	74.15
DR-GCN	81.47	80.89	72.36	76.39
ContractGIB	**84.72**	**82.95**	**74.31**	**78.40**

Table 4: Performance comparison on Infinite Loop vulnerability detection. The best performance is in bold

Methods	Acc (%)	Recall (%)	Precision (%)	F1 (%)
Jolt	42.88	23.11	38.23	28.81
PDA	46.44	21.73	42.96	28.26
SMT	54.04	39.23	55.69	45.98
Looper	59.56	47.21	62.72	53.87
Vanilla-RNN	49.57	47.86	42.10	44.79
LSTM	51.28	57.26	44.07	49.80
GRU	51.70	50.42	45.00	47.55
GCN	64.01	63.04	59.96	61.46
DR-GCN	68.34	67.82	64.89	66.32
ContractGIB	**74.82**	**74.32**	**69.89**	**69.12**

4.3 Interpretability Analysis

This section illustrates the complete workflow of our framework, from transforming raw Solidity source code into a compact contract graph, to vulnerability prediction, and finally to explanation generation. Two real-world contracts are used as demonstrative case studies: `MultiSigWallet.executeTransaction` (Fig. 3) and `Accrual_account.FundTransfer` (Fig. 4). Despite their differences in code structure and complexity, both examples are processed through an identical parsing, graph construction, and explanation pipeline.

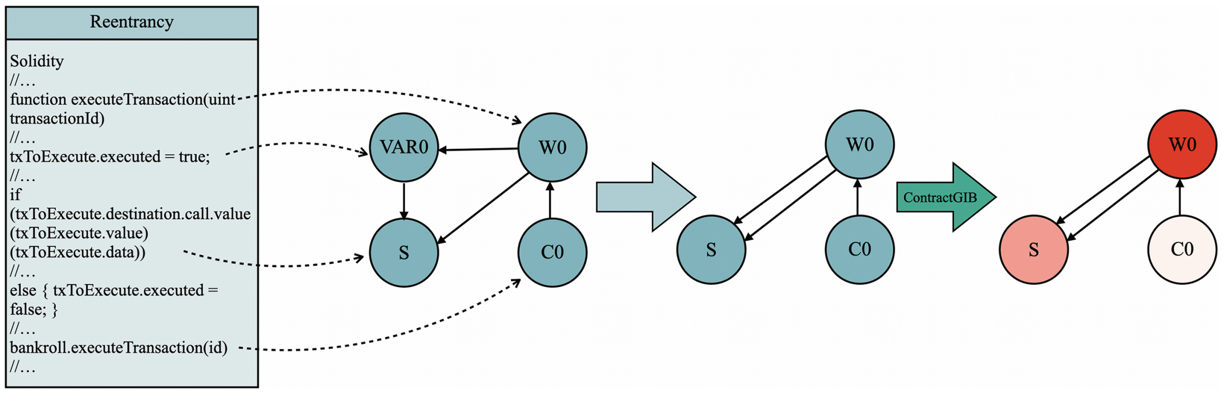
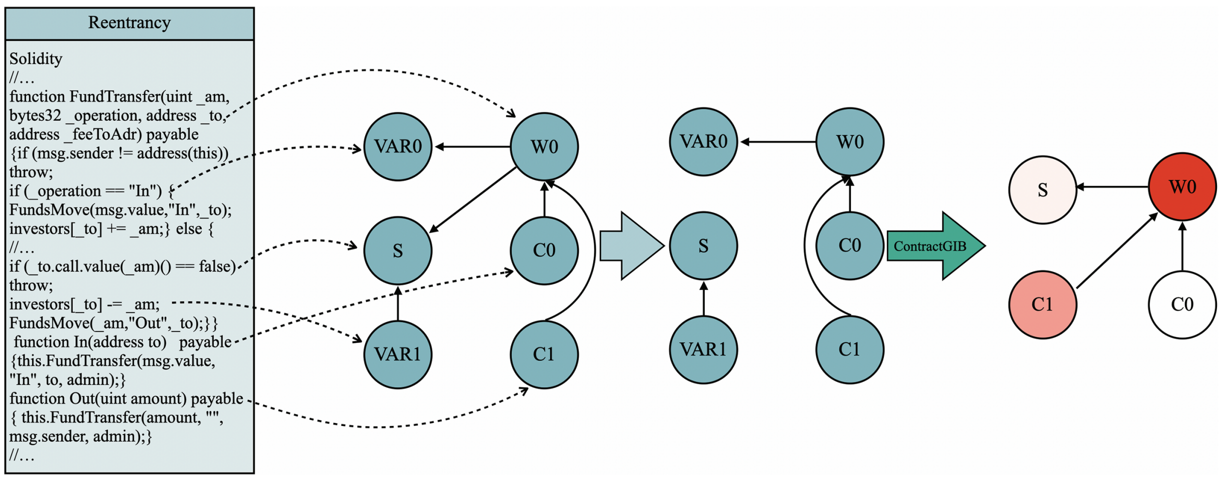

Reentrancy
Solidity
//...
function executeTransaction(uint transactionId)
//...
txToExecute.executed = true;
//...
if
(txToExecute.destination.call.value
(txToExecute.value)
(txToExecute.data))
//...
else { txToExecute.executed = false; }
//...
bankroll.executeTransaction(id)
//...
VAR0
W0
S
C0
ContractGIB

Figure 3: Extraction and attribution for `executeTransaction`

Example 1: `executeTransaction`.

The static scanner retains only four statements that are essential for characterizing the control logic of the function: (i) the function signature, (ii) the assignment that updates the execution lock `txToExecute.executed = true`, (iii) the outbound low-level call `destination.call.value(...)(...)`, and (iv) the rollback assignment in the `else` branch.

These statements are abstracted into semantic nodes representing a synthetic caller (C0), the function body (W0), the storage-backed lock variable, and the external call site (S). A minimal control-data dependency chain arises from this abstraction, captured through three edges (FW, GN, IF), yielding the core subgraph C0 → W0 → S.



```
Solidity
//…
function FundTransfer(uint _am,
bytes32 _operation, address _to,
address _feeToAdr) payable
{if (msg.sender != address(this))
throw;
if (_operation == "In") {
FundsMove(msg.value,"In",_to);
investors[_to] += _am;} else {
//…
if (_to.call.value(_am)() == false)
throw;
investors[_to] -= _am;
FundsMove(_am,"Out",_to);}}
 function In(address to)   payable
{this.FundTransfer(msg.value,
"In", to, admin);}
function Out(uint amount) payable
{ this.FundTransfer(amount, "",
msg.sender, admin);}
//…
```

Figure 4: Extraction and attribution for `Accrual_account.FundTransfer`

After one-hot encoding the nodes (250 dimensions each), the classifier operates on this compact three-node graph. Attribution analysis using Integrated Gradients consistently assigns the highest relevance to W0, followed by S, with C0 contributing only marginally. This ranking aligns with expert reasoning: the function body `executeTransaction` governs the ordering of the two critical steps that determine reentrancy safety—the lock assignment and the subsequent external call. Whether the pattern "write lock before call" is preserved is both necessary and sufficient for deciding if the function is protected. The call site S only indicates that an external transfer occurs, while the synthetic caller C0 does not influence the internal sequencing. Thus, both auditors and the attribution model arrive at the same causal ordering, W0 > S $\gg$ C0, confirming that the function body carries the decisive semantic signal for assessing vulnerability.

Example 2: `FundTransfer`.

Here the scanner retains five fragments (line numbers in Listing 4-B).

The static analysis stage extracts only a small set of code fragments from `FundTransfer` that are relevant to its behavioral structure. The function declaration at line 9 is abstracted into the function node W0. The branch condition `_operation=="In"` on line 13 is recorded as an auxiliary predicate node VAR0, although this node is later removed during graph compression. The low-level external invocation `_to.call.value(_am)()` on line 18 is mapped to the call-site node S. Immediately after the call, the balance update `investors [_to]-=_am` on line 19 becomes node VAR1, reflecting the characteristic *state-update-after-interaction* pattern. Finally, the two wrapper functions declared in lines 24–30 (`In` and `Out`), both of which delegate to `this.FundTransfer(...)`, are represented by the synthetic entry nodes C0 and C1, providing the external entry points into the core logic.

From these fragments, the scanner constructs the control/data-flow edges C0 → W0, C1 → W0, VAR1 → S (capturing the effect-after-call dependency), and W0 → VAR0 (recording the predicate read). Since VAR0 is used only once and carries no independent semantic role, the compression step removes it, yielding a four-node graph of the form $\{C0, C1\} \rightarrow W0 \rightarrow S$.

Integrated Gradients applied to this reduced structure produces the ranking W0 > C1 > S > C0. The model assigns highest relevance to W0, consistent with the fact that the function body governs how state is updated and whether an unsafe call sequence may occur. The `Out` wrapper (C1) receives the next-highest weight because its execution path reaches the low-level call without any locking mechanism. The call-site node S serves primarily as supporting evidence, indicating that an external transfer takes place. The synthetic `In` entry (C0), which never invokes the dangerous interaction, contributes minimally.

In both contract examples, the explanatory attributions isolate precisely the elements that a human auditor would identify as security-critical, while nodes introduced solely for connectivity are down-weighted. This alignment between automated attribution and expert reasoning demonstrates that the graph reduction procedure preserves all vulnerability-relevant semantics, and that the interpretability module exposes the true causal structure underlying the model's predictions.

4.4 Ablation Study

To better understand the contribution of each branch in our dual-branch encoder, we conduct an ablation study by isolating the structural and semantic components. In the *Structure-only* variant, the model processes node representations solely through the structural GNN branch, effectively removing all contextual information derived from CodeBERT. Conversely, the *Sem-only* variant retains only the LLM-enhanced semantic branch, while discarding the structural propagation mechanism. Comparing these two variants allows us to assess the relative importance of explicit graph-structural dependencies *versus* learned semantic cues. The results (Tables 5 and 6) show that each branch captures complementary information: the structural branch excels at modeling control-flow and data-flow patterns, whereas the semantic branch contributes richer contextual understanding of the underlying code semantics. The full model, which jointly integrates both branches, achieves the best performance, confirming that structural relations and LLM-derived semantics are mutually reinforcing for accurate vulnerability detection.

Table 5: Ablation study of encoder components on Reentrancy vulnerability detection. The best performance is in bold

Variant	Acc (%)	Recall (%)	Precision (%)	F1 (%)
Structure-only	81.10	79.45	71.20	75.11
Sem-only	82.01	80.02	72.10	75.88
Full model (ContractGIB)	**84.72**	**82.95**	**74.31**	**78.40**

Table 6: Ablation study of encoder components on Infinite Loop vulnerability detection. The best performance is in bold

Variant	Acc (%)	Recall (%)	Precision (%)	F1 (%)
Structure-only	72.95	72.10	67.45	69.70
Sem-only	73.52	73.01	68.12	70.47
Full model (ContractGIB)	**74.82**	**74.32**	**69.89**	**69.12**

Furthermore, Fig. 5 illustrates that our use of HSIC significantly improves training stability compared to both the variational approximation method (MINE) and the matrix-based entropy functional. We also evaluate the computational efficiency of ContractGIB. In practice, HSIC-based dependence estimation is both faster and more stable than variational mutual-information estimators such as MINE, which require training an auxiliary network and introduce additional optimization variance. On both ESC and VSC, HSIC adds negligible overhead and allows ContractGIB to maintain a lightweight training loop. Empirically, ContractGIB trains approximately 1.3–1.5× faster per epoch compared to models that rely on variational MI estimators, largely because HSIC involves only kernel computations rather than adversarial optimization.

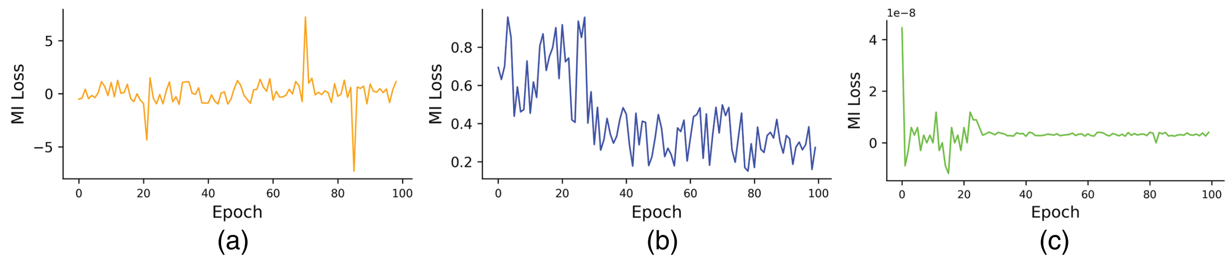


Figure 5: Curves of $I(G; G_{\text{sub}})$ over epochs on reentrancy vulnerability detection. (**a**) $I(G; G_{\text{sub}})$ estimated with the variational mutual information neural estimator (MINE), showing large fluctuations/instability across epochs. (**b**) $I(G; G_{\text{sub}})$ estimated with the matrix-based Rényi's α-order mutual-information estimator, which remains noisy and less stable during optimization. (**c**) $I(G; G_{\text{sub}})$ estimated with the proposed HSIC-based dependence measure, exhibiting a smoother, lower-variance trajectory and more stable convergence

We then quantify the contribution of the proposed LLM components by comparing (i) *handcrafted* node features used in prior work, and (ii) *CodeBERT-augmented* node features proposed in Sections 3.2 and 3.3.2.

Compared variants.

A1 **Handcrafted:** One-hots and structural attributes only (node type, degree, control-/data-flow flags), no LLM embeddings.

A2 **CodeBERT-Init:** Initialize node features with $x_i^{(0)} = f_{\text{CodeBERT}}(c_i)$; no extra concatenation.

A3 **CodeBERT-Ann:** Concatenate semantic annotation $e_i = f_{\text{CodeBERT}}(c_i)$ to handcrafted features: $\tilde{x}_i = [x_i \oplus e_i]$.

A4 **CodeBERT-Init+Ann** (full): Initialization + concatenation (our default).

All variants share the same dual-branch encoder, subgraph generator, HSIC objective (Eq. (18)), optimizer, and training schedule. We tune λ, β on the validation set for each variant to ensure fairness.

We report Accuracy, Precision, Recall, and F1 on the test split (20%) in Tables 7 (Reentrancy/ESC) and 8 (Infinite Loop/VSC). Each experiment is repeated with 5 random seeds; we report mean ± std and conduct paired t-tests against A1 (Handcrafted). Statistical significance is marked with † ($p < 0.05$) and ‡ ($p < 0.01$).

Table 7: Ablation on feature design for Reentrancy (ESC). Results (%) are mean ± std over 5 runs. A4 corresponds to ContractGIB (see Table 3). The best performance is in bold. Statistical significance is marked with † ($p < 0.05$) and ‡ ($p < 0.01$)

Variant	Acc	Prec	Rec	F1
A1 Handcrafted	81.2 ± 0.5	72.1 ± 0.7	80.3 ± 0.6	76.0 ± 0.5
A2 CodeBERT-Init	82.7 ± 0.4†	73.6 ± 0.6	81.0 ± 0.5	77.0 ± 0.4†
A3 CodeBERT-Ann	83.9 ± 0.4‡	74.5 ± 0.5‡	82.3 ± 0.5‡	78.0 ± 0.4‡
A4 Init+Ann (ContractGIB)	**84.72 ± 0.3**‡	**74.31 ± 0.4**‡	**82.95 ± 0.4**‡	**78.40 ± 0.3**‡

We observed that: (i) CodeBERT-based initialization (A2) already yields consistent gains over handcrafted features (A1), indicating that pretrained code semantics provide a stronger starting point. (ii) Semantic annotation via concatenation (A3) improves further, suggesting complementary benefits between structural attributes and LLM-derived context. (iii) Combining both (A4) achieves the same results as our full ContractGIB model, confirming that LLM-enhanced features are critical for state-of-the-art performance.

Table 8: Ablation on feature design for Infinite Loop (VSC). Results (%) are mean ± std over 5 runs. A4 corresponds to ContractGIB (see Table 4). The best performance is in bold. Statistical significance is marked with † ($p < 0.05$) and ‡ ($p < 0.01$)

Variant	Acc	Prec	Rec	F1
A1 Handcrafted	68.1 ± 0.6	64.8 ± 0.7	67.3 ± 0.8	66.0 ± 0.6
A2 CodeBERT-Init	70.4 ± 0.5†	67.9 ± 0.6†	70.5 ± 0.6†	69.1 ± 0.5†
A3 CodeBERT-Ann	71.8 ± 0.4‡	69.2 ± 0.5‡	72.0 ± 0.5‡	70.5 ± 0.4‡
A4 Init+Ann (ContractGIB)	**74.82 ± 0.3**‡	**69.89 ± 0.4**‡	**74.32 ± 0.4**‡	**69.12 ± 0.3**‡

The ablation study isolates the contribution of each module: A1 shows the effect of removing HSIC-based subgraph selection, A2 assesses the impact of dropping the GIB regularizer, A3 evaluates the contribution of CodeBERT embeddings, and A4 removes the subgraph generator entirely. The performance drop across A1–A4 highlights the complementary roles of semantic encoding, HSIC-guided compression, and structural subgraphs.

5 Conclusions

In this work, we proposed **ContractGIB**, a novel framework for interpretable smart contract vulnerability detection that leverages the information bottleneck (IB) principle within a graph neural network (GNN) architecture. To realize this framework, we introduced three key innovations. First, we replaced unstable variational mutual information estimators with the *Hilbert–Schmidt Independence Criterion* (HSIC), providing a principled, kernel-based dependence measure that ensures stable and efficient optimization. Second, we incorporated a *CodeBERT-assisted semantic annotation* module to enrich node features with vulnerability-sensitive embeddings, allowing the model to capture subtle contextual cues in smart contract code. Third, we employed a *CodeBERT-based initialization* strategy that replaces handcrafted encodings with pretrained code embeddings, improving convergence and robustness against obfuscation. We evaluated ContractGIB on both reentrancy and infinite loop detection tasks using datasets from Ethereum and the VNT chain. Across comparisons with 12 existing methods, ContractGIB consistently achieves superior detection accuracy and interpretability, offering a practical and trustworthy solution for smart contract security auditing.

While ContractGIB provides strong performance and interpretable subgraph explanations, the framework depends on CodeBERT's pretrained representations and may face challenges when applied to contracts written in languages or styles not captured in the pretraining *corpus*. Scaling the method to very large monolithic contracts or multi-file distributed project structures is an important direction for future work.

Acknowledgement: We acknowledge the maintainers of the public ESC and VSC benchmark datasets and related open-source tooling that supported our experimental evaluation.

Funding Statement: This work was supported by the National Natural Science Foundation of China (Grant Nos. 52208424, 52208416, 52078091, and 52108399) and the Shanghai Municipal Science and Technology Major Project (Grant No. 2021SHZDZX0102).

Author Contributions: Conceptualization, Yiming Yu and Junliang Du; methodology, Yiming Yu and Junliang Du; software, Yiming Yu, Yunfei Guo and Junchen Liu; validation, Yiming Yu, Yunfei Guo and Junchen Liu; formal analysis, Yiming Yu; investigation, Yiming Yu; writing—original draft preparation, Yiming Yu; writing—review and editing, Junliang Du; visualization, Yiming Yu, Yunfei Guo and Junchen Liu; supervision, Yiping Sun and Junliang Du; funding acquisition, Junliang Du. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The ESC and VSC benchmark datasets analyzed in this study are publicly available from the GNNSCVulDetector repository at https://github.com/vntchain/GNNSCVulDetector/tree/master.

Ethics Approval: This study did not involve human participants or animals. Ethical approval was therefore not required.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

Abbreviations

The following abbreviations are used in this manuscript:

IB	Information Bottleneck
GIB	Graph Information Bottleneck
HSIC	Hilbert-Schmidt Independence Criterion

References

1. Kiani R, Sheng VS. Ethereum smart contract vulnerability detection and machine learning-driven solutions: a systematic literature review. Electronics. 2024;13(12):2295. doi:10.3390/electronics13122295.
2. Sürücü O, Yeprem U, Wilkinson C, Hilal W, Gadsden SA, Yawney J, et al. A survey on ethereum smart contract vulnerability detection using machine learning. In: Disruptive technologies in information sciences VI. Vol. 12117. 2022. p. 110–21.
3. Mueller B. Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amst. 2018;9(54):4–17.
4. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM; 2016. p. 254–69.
5. Sun Y, Gu L. Attention-based machine learning model for smart contract vulnerability detection. J Phys Conf Ser. 2021;1820:012004. doi:10.1088/1742-6596/1820/1/012004.
6. Zhou Q, Zheng K, Zhang K, Hou L, Wang X. Vulnerability analysis of smart contract for blockchain-based IoT applications: a machine learning approach. IEEE Internet Things J. 2022;9(24):24695–707. doi:10.1109/jiot.2022.3196269.
7. Tang X, Du Y, Lai A, Zhang Z, Shi L. Deep learning-based solution for smart contract vulnerabilities detection. Sci Rep. 2023;13(1):20106. doi:10.1038/s41598-023-47219-0.
8. Zhang L, Wang J, Wang W, Jin Z, Zhao C, Cai Z, et al. A novel smart contract vulnerability detection method based on information graph and ensemble learning. Sensors. 2022;22(9):3581.
9. Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. arXiv:1609.02907. 2017.
10. Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q. Smart contract vulnerability detection using graph neural networks. In: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence; 2021 Jan 7–15; Yokohama, Japan. p. 3283–90.
11. Wang Y, Zhao X, He L, Zhen Z, Chen H. ContractGNN: ethereum smart contract vulnerability detection based on vulnerability sub-graphs and graph neural networks. IEEE Trans Netw Sci Eng. 2024;11(6):6382–95. doi:10.1109/tnse.2024.3470788.
12. Lutz O, Chen H, Fereidooni H, Sendner C, Dmitrienko A, Sadeghi AR, et al. ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. arXiv:2103.12607. 2021.
13. Zeng Q, He J, Zhao G, Li S, Yang J, Tang H, et al. EtherGIS: a vulnerability detection framework for ethereum smart contracts based on graph learning features. In: 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC). Piscataway, NJ, USA: IEEE; 2022. p. 1742–9.
14. Fadi O, Bahaj A, Zkik K, El Ghazi A, Ghogho M, Boulmalf M. Smart contract anomaly detection: the contrastive learning paradigm. Comput Netw. 2025;260(1):111121. doi:10.1016/j.comnet.2025.111121.
15. Xu C, Xu H, Zhu L, Shen X, Sharif K. Enhanced smart contract vulnerability detection via graph neural networks: achieving high accuracy and efficiency. IEEE Trans Softw Eng. 2025;51(6):1854–65. doi:10.1109/tse.2025.3570421.
16. Tishby N, Pereira FC, Bialek W. The information bottleneck method. arXiv:physics/0004057. 2000.

17. Yu J, Xu T, Rong Y, Bian Y, Huang J, He R. Recognizing predictive substructures with subgraph information bottleneck. IEEE Trans Pattern Anal Mach Intell. 2021;46(3):1650–63. doi:10.1109/tpami.2021.3112205.
18. Belghazi MI, Baratin A, Rajeshwar S, Ozair S, Bengio Y, Courville A, et al. Mutual information neural estimation. In: International Conference on Machine Learning. London, UK: PMLR; 2018. p. 531–40.
19. Yu S, Giraldo LGS, Jenssen R, Principe JC. Multivariate extension of matrix-based renyi's α-order entropy functional. IEEE Trans Pattern Anal Mach Intell. 2019;42(11):2960–6. doi:10.1109/tpami.2019.2932976.
20. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. CodeBERT: a pre-trained model for programming and natural languages. In: Findings of the association for computational linguistics: EMNLP 2020. Stroudsburg, PA, USA: ACL; 2020. p. 1536–47.
21. Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Buenzli F, Vechev M. Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM; 2018. p. 67–82.
22. Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. New York, NY, USA: ACM; 2020. p. 778–88.
23. Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK. Smartian: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway, NJ, USA: IEEE; 2021. p. 227–39.
24. Contro F, Crosara M, Ceccato M, Dalla Preda M. EtherSolve: computing an accurate control-flow graph from ethereum bytecode. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). Piscataway, NJ, USA: IEEE; 2021. p. 127–37.
25. Arceri V, Merenda SM, Dolcetti G, Negrini L, Olivieri L, Zaffanella E. Towards a sound construction of evm bytecode control-flow graphs. In: Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs. New York, NY, USA: ACM; 2024. p. 11–6.
26. Grech N, Brent L, Scholz B, Smaragdakis Y. Gigahorse: thorough, declarative decompilation of smart contracts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Piscataway, NJ, USA: IEEE; 2019. p. 1176–86.
27. Albert E, Gordillo P, Livshits B, Rubio A, Sergey I. EthiR: a framework for high-level analysis of ethereum bytecode. In: International Symposium on Automated Technology for Verification and Analysis. Cham, Switzerland: Springer; 2018. p. 513–20.
28. Schneidewind C, Grishchenko I, Scherer M, Maffei M. eThor: practical and provably sound static analysis of ethereum smart contracts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM; 2020. p. 621–40.
29. Albert E, Grossman S, Rinetzky N, Rodríguez-Núñez C, Rubio A, Sagiv M. Taming callbacks for smart contract modularity. Proc ACM Program Lang. 2020;4(OOPSLA):1–30.
30. Gogineni AK, Swayamjyoti S, Sahoo D, Sahu KK, Kishore R. Multi-class classification of vulnerabilities in smart contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing. IOP SciNotes. 2020;1(3):035002. doi:10.1088/2633-1357/abcd29.
31. Liao JW, Tsai TT, He CK, Tien CW. SoliAudit: smart contract vulnerability assessment based on machine learning and fuzz testing. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). Piscataway, NJ, USA: IEEE; 2019. p. 458–65.
32. Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. arXiv:1301.3781. 2013.
33. Ashizawa N, Yanai N, Cruz JP, Okamura S. Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure. New York, NY, USA: ACM; 2021. p. 47–59.
34. Liu Z, Qian P, Wang X, Zhu L, He Q, Ji S. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. arXiv:2106.09282. 2021.

35. Boi B, Esposito C, Lee S. Smart contract vulnerability detection: the role of large language model (LLM). ACM SIGAPP Appl Comput Rev. 2024;24(2):19–29. doi:10.1145/3687251.3687253.
36. Tamberg K, Bahsi H. Harnessing large language models for software vulnerability detection: a comprehensive benchmarking study. IEEE Access. 2025;13(10):29698–717. doi:10.1109/access.2025.3541146.
37. Gilad-Bachrach R, Navot A, Tishby N. An information theoretic tradeoff between complexity and accuracy. In: Learning Theory and Kernel Machines: 16th Annual Conference on Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003; 2003 Aug 24–27; Washington, DC, USA. Cham, Switzerland: Springer; 2003. p. 595–609.
38. Wu T, Ren H, Li P, Leskovec J. Graph information bottleneck. Adv Neural Inf Process Syst. 2020;33:20437–48.
39. Sun H, Pears N, Gu Y. Information bottlenecked variational autoencoder for disentangled 3d facial expression modelling. In: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision. Piscataway, NJ, USA: IEEE; 2022. p. 157–66.
40. Zhai P, Zhang S. Adversarial information bottleneck. IEEE Trans Neural Netw Learn Syst. 2022;35(1):221–30. doi:10.1109/tnnls.2022.3172986.
41. Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. arXiv:1711.00740. 2018.
42. Gretton A, Fukumizu K, Teo C, Song L, Schölkopf B, Smola A. A kernel statistical test of independence. In: Proceedings of the 21st International Conference on Neural Information Processing Systems; 2007 Dec 3–6; Vancouver, BC, Canada. Red Hook, NY, USA: Curran Associates Inc; 2007. p. 585–92.
43. Wang Z, Jian T, Masoomi A, Ioannidis S, Dy J. Revisiting hilbert-schmidt information bottleneck for adversarial robustness. Adv Neural Inf Process Syst. 2021;34:586–97.
44. Dong F, Jieren C, Xiangyan T, Kai L. Smart contract vulnerability detection method based on node feature augmentation graph convolutional network. In: International Conference on Information Science, Communication and Computing. Cham, Switzerland: Springer; 2024. p. 102–19.
45. Gupta NA, Bansal M, Sharma S, Mehrotra D, Kakkar M. Detection of vulnerabilities in blockchain smart contracts using deep learning. Wirel Netw. 2025;31(1):201–17. doi:10.1007/s11276-024-03755-9.
46. Ferreira JF, Cruz P, Durieux T, Abreu R. SmartBugs: a framework to analyze solidity smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Piscataway, NJ, USA: IEEE; 2020. p. 1349–52.
47. Ghaleb A, Pattabiraman K. 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. New York, NY, USA: ACM; 2020. p. 415–27.
48. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. SmartCheck: static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. New York, NY, USA: ACM; 2018. p. 9–16.
49. Carbin M, Misailovic S, Kling M, Rinard MC. Detecting and escaping infinite loops with jolt. In: European Conference on Object-Oriented Programming. Cham, Switzerland: Springer; 2011. p. 609–33.
50. Ibing A, Mai A. A fixed-point algorithm for automated static detection of infinite loops. In: 2015 IEEE 16th International Symposium on High Assurance Systems Engineering. Piscataway, NJ, USA: IEEE; 2015. p. 44–51.
51. Kling M, Misailovic S, Carbin M, Rinard M. Bolt: on-demand infinite loop escape in unmodified binaries. ACM SIGPLAN Not. 2012;47(10):431–50.
52. Burnim J, Jalbert N, Stergiou C, Sen K. Looper: lightweight detection of infinite loops at runtime. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. Piscataway, NJ, USA: IEEE; 2009. p. 161–9.
53. Yang J, Liu S, Dai S, Fang Y, Xie K, Lu Y. ByteEye: a smart contract vulnerability detection framework at bytecode level with graph neural networks. Autom Softw Eng. 2026;33(1):1–38. doi:10.1007/s10515-025-00559-9.
54. Ma F, Xu Z, Ren M, Yin Z, Chen Y, Qiao L, et al. Pluto: exposing vulnerabilities in inter-contract scenarios. IEEE Trans Softw Eng. 2021;48(11):4380–96. doi:10.1109/tse.2021.3117966.