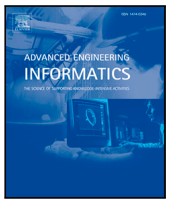
Advanced Engineering Informatics 74 (2026) 104701

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei



Full length article

# VulGNN: A high-fidelity graph neural network framework for robust smart contract vulnerability detection

Xinyi Chen [a,1], Weihua Bai [a,1], Jialing Zhao [b,*], Tao Huang [a], Huibing Zhang [c], Teng Zhou [d,e,**], Keqin Li [f]

[a] School of Computer Science and Software, Zhaoqing University, Zhaoqing, Guangdong 526061, China
[b] Modern Educational Technology Center, Zhaoqing University, Zhaoqing, Guangdong 526061, China
[c] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China
[d] School of Cyberspace Security, Hainan University, Haikou 570228, China
[e] Yangtze Delta Region Institute, University of Electronic Science and Technology of China, Quzhou 324003, China
[f] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

*Keywords:*
Smart contract security
Graph neural network
Ensemble learning
Data curation

ABSTRACT

Smart contracts are fundamental components of blockchain systems that execute digital asset transfers and protocol logic. Their security directly impacts the reliability and stability of the blockchain ecosystem. Recently, graph neural network (GNN)-based approaches have shown promise for vulnerability detection due to their ability to model structured code. However, progress is often limited by noisy and structurally inconsistent public datasets as well as implementation complexity that hinders reproducibility. This work presents a high-fidelity and scalable pipeline for smart contract vulnerability detection. The pipeline integrates an automated data curation procedure that constructs clean, balanced, and structurally consistent datasets from verified Etherscan source code. On top of the curated graphs, we develop a dual-channel architecture that models topology message passing and traversal-derived edge-path semantics in parallel to produce expressive code representations, termed VulGNN-SG. Implemented on PyTorch Geometric, VulGNN-SG achieves an F1-score of 92.69% on benchmark datasets under a consistent evaluation protocol. Extensive experiments provide controlled empirical evidence that graph traversal strategies are strongly associated with the semantic complexity of vulnerability patterns. To improve robustness and generalization, we further adopt a $K$-fold ensemble variant (VulGNN-SG+), which consistently improves performance across four representative vulnerability types, with a maximum gain of 3.06%. Overall, this work provides a reproducible end-to-end pipeline for smart-contract vulnerability detection and offers controlled empirical evidence on how data fidelity and traversal-induced semantics affect detection robustness. In addition to smart-contract security as the evaluation domain, we position the contribution as an artifact-centered assurance workflow component that automates knowledge-intensive inspection of executable digital artifacts by providing reproducible compilation-verified curation, dependency-preserving representations, and traceable screening evidence.

## 1. Introduction and motivation

Smart contracts are executable software artifacts that implement predefined logic and govern digital assets in blockchain-based systems, thereby enabling decentralized finance (DeFi) and a broad range of data-driven applications. However, once deployed, smart contracts are effectively immutable: vulnerabilities caused by coding errors cannot be patched in place and may trigger irreversible financial loss. Incidents such as the DAO exploit have demonstrated the high stakes of pre-deployment assurance, motivating the development of reliable and automated techniques for vulnerability detection and security verification [1]. In practice, such techniques support knowledge-intensive tasks, such as audit triage, compliance-oriented review, and release gating in continuous integration pipelines, by prioritizing suspicious contracts and reducing manual workload. From an engineering-informatics perspective, this workflow is an artifact-centered assurance process

* Corresponding author.
** Corresponding author at: School of Cyberspace Security, Hainan University, Haikou 570228, China.
*E-mail addresses:* chenxinyi@stu.zqu.edu.cn (X. Chen), baiweihua@zqu.edu.cn (W. Bai), zhaojialing@zqu.edu.cn (J. Zhao), teng.zhou@hainanu.edu.cn (T. Zhou).
[1] Xinyi Chen and Weihua Bai contributed equally to this work.

https://doi.org/10.1016/j.aei.2026.104701
Received 10 November 2025; Received in revised form 3 March 2026; Accepted 12 April 2026
Available online 25 April 2026
1474-0346/

for engineered software artifacts. Practitioners must inspect complex design and implementation decisions under time and cost constraints, and automated screening systems are used to support decision-making by providing consistent risk signals and traceable evidence. Accordingly, we formulate the problem as automating a knowledge-intensive inspection task on executable software artifacts, where structured representations (syntax, control flow, and data flow) are required to capture non-local dependencies.

Early vulnerability detection methods primarily relied on program analysis techniques such as symbolic execution, static analysis, and rule-based checkers. While effective for well-specified patterns, these approaches often face path explosion, incomplete rule coverage, and brittle handling of inter-procedural dependencies, which collectively limit scalability and robustness on heterogeneous real-world contracts. In parallel, learning-based approaches have been explored to complement rule-driven reasoning. Sequence-based models can capture lexical regularities but do not explicitly encode control- and data-dependence, which are frequently essential to characterize vulnerability-triggering conditions (e.g., whether a return value is checked along a feasible execution path, or whether a critical variable is defined and used across basic blocks). This observation motivates graph-based representations, where abstract syntax, control flow, and data flow can be unified into a semantic graph, enabling relational reasoning over non-local dependencies. While sequence models can still be effective when vulnerabilities manifest as local lexical cues or when large labeled corpora are available, many security-relevant conditions are inherently relational (e.g., def-use chains and path-sensitive checks). This makes graph representations a natural choice for our setting, where such dependencies must be modeled in a unified structure. Accordingly, graph neural networks (GNNs) are well-suited for smart contract analysis because they operate directly on structured program graphs and can integrate both topology and attributed semantics [2].

Despite the promising results reported by state-of-the-art (SOTA) models, their effectiveness under realistic conditions is not yet well established. A persistent gap exists between controlled benchmark performance and deployment-oriented robustness on diverse public code corpora. We identify three recurring failure modes that complicate reliable evaluation and hinder fair comparisons across methods. (1) *Non-compilable and structurally inconsistent corpora.* Public repositories such as smartbugs-wild [3] aggregate contracts from diverse sources. Variations in compiler versions and language features can make a substantial portion of samples incompatible with a unified toolchain, leading to unstable parsing and graph construction; subsequent filtering may introduce evaluation bias and obscure true generalization. (2) *Engineering bottlenecks in data preprocessing.* Many open-source implementations adopt Just-in-Time (JIT) graph construction and loading during training to simplify the pipeline. However, our reproduction indicates that this convenience can become a dominant I/O bottleneck at scale: repeated parsing and graph building across epochs substantially increase end-to-end runtime, extending a single experimental run from minutes to more than 10 h on large corpora. This overhead makes large-scale cross-validation and multi-task evaluations difficult in practice, thereby limiting both the computational efficiency and the scalability of rigorous model assessment. (3) *Methodological risks in evaluation protocols.* Repeated random sub-sampling and insufficient de-duplication can place near-duplicate or source-related contracts across training and test splits, causing information leakage and optimistic generalization estimates.

In an artifact-centered assurance workflow, the practical "why" is to reduce the cognitive and time burden of pre-deployment inspection while preserving decision accountability: reviewers must quickly determine which artifacts merit attention and why, under strict constraints on compilation/toolchain stability and evaluation reliability. The "how" in this paper is a design-by-failure-mode synthesis: we start from recurring breakdowns observed in public smart-contract corpora and in prior reproducibility attempts, and we map each breakdown to a workflow design decision and to empirical evidence reported in later sections.

Concretely, the framework is organized as a traceable chain from engineering task requirements to validated pipeline choices:

FM1: Non-compilable or structurally inconsistent artifacts → compilation-verified data curation and structural validation (Sections 3.3–3.4) → evidence: curated coverage over smartbugs-wild (76.16%) and the corresponding exclusion rate ($\tilde{2}$3.84%) due to compilation/logic checks; final curated dataset scale and balance (Table 1) and the validated unique corpus used for SG generation (Section 3.4).

FM2: JIT preprocessing as a dominant I/O bottleneck → ahead-of-time (AOT) preprocessing that decouples graph construction/feature generation from training (Section 3.5.1) → evidence: serial 5-fold training becomes practically feasible (Table 9 reports 2.5 h total training on a representative task; inference overhead for the ensemble is explicitly quantified).

FM3: Split leakage and unreliable evaluation protocols → fixed split policy, file-level de-duplication, 5-fold cross-validation with a held-out test set, and unified re-evaluation of baselines (Section 4.1.2–4.2) → evidence: all methods are compared under identical inputs/splits and a controlled protocol (Tables 6–7), and robustness gains from the fold-reuse ensemble are quantified (Table 8).

Why traversal-aware dual-channel modeling is not a "component stack" → separating topology-driven dependency aggregation from traversal-derived edge-path semantics (Section 3.5–5.2.3) → evidence: BFS/DFS effects differ systematically by vulnerability category under the same protocol, and the mechanistic rationale is grounded in dependency-chain examples (Section 3.5.1; Figs. 4–5; Section 5.2.3).

Beyond data and evaluation challenges, model design should be justified in terms of the information required for vulnerability detection. Many GNN-based detectors (e.g., Devign) adopt monolithic architectures that fuse heterogeneous code signals into a single graph encoder, which can entangle global structural context with localized semantic cues and make it difficult to isolate the dependency patterns that drive prediction [4]. In contrast, vulnerability evidence may manifest both as global dependence structure (e.g., control/data-flow connectivity) and as ordered semantic cues along execution-related paths [5]. This motivates an explicit separation of concerns in the representation learning pipeline.

To address these challenges and facilitate reproducible experimentation for learning-based vulnerability detection, we propose VulGNN, a dual-channel graph neural network framework for smart contract security analysis.

Although our experiments focus on smart contracts, the proposed pipeline addresses a broader class of knowledge-intensive engineering tasks on digital artifacts by automating program understanding, quality assurance, and risk identification through data-centric and graph-based learning. Specifically, the proposed data curation and graph-learning pipeline is designed as an engineering workflow component for pre-deployment assurance, where artifacts are compiled, structurally validated, represented as multi-relational semantic graphs, and then screened to prioritize expert review. The methodology is broadly relevant to artifact-centered software engineering scenarios in which defects or policy violations are characterized by structured dependencies (e.g., control/data-flow relations), even though we do not claim immediate empirical generalization beyond the studied contract datasets. We emphasize that the empirical validation in this paper is confined to smart-contract artifacts; extending and validating the workflow on other artifact-centered engineering domains requires domain-specific toolchains and datasets and is therefore left as future work.

The proposed framework contributes an auditable data/pipeline design and a controlled traversal study. Our contribution is an architecture-level integration and the associated preprocessing protocol: (i) we construct a multi-relational semantic graph tailored to smart-contract semantics; (ii) we decouple representation learning into

a topology channel and a traversal-derived edge-path semantic channel to avoid conflating global structure with ordered semantic evidence; and (iii) We enable controlled analysis of BFS/DFS as a modeling factor rather than an incidental implementation choice. The ablation study empirically validates these design choices and traversal analysis reported in Section 5.

VulGNN decouples representation learning into two specialized branches: a deep residual GCN-based structural module that captures global topological dependencies in the semantic graph, and a Transformer-based edge module that learns contextual semantics along traversal-derived paths. By separately modeling structural dependencies and traversal-sensitive semantics, VulGNN supports complementary feature learning and enables controlled analysis of preprocessing and evaluation choices.

In the targeted assurance setting, the primary knowledge-intensive engineering tasks are pre-deployment audit triage, release gating in continuous integration, and compliance-oriented review, where the artifact under inspection must be compiled, structurally validated, and screened with traceable evidence to support time-bounded expert decisions. In the context of blockchain-enabled engineering systems, smart contracts function as executable digital artifacts that enforce process logic – such as automated payment settlement, escrow release, and access-control gating – on behalf of engineering workflows (cf. Fig. 17).

Building on the above observations, this work pursues the following research objectives. **O1 (Artifact fidelity and decision reliability):** to construct a compilation-verified and structurally consistent dataset from Etherscan-verified contract artifacts, together with a controlled preprocessing pipeline that reduces parsing-induced noise and enables reproducible evaluation. **O2 (Task effectiveness for automated triage):** to assess whether a dual-channel encoder that separates topology-level dependency modeling from traversal-derived edge-path semantics improves screening performance across multiple vulnerability categories. **O3 (Traversal-derived evidence semantics):** to quantify how BFS vs. DFS traversal affects detection performance by vulnerability type under an identical protocol, and to provide a mechanistic interpretation grounded in dependency structure. **O4 (Workflow robustness under fixed protocols):** to evaluate whether a fold-reuse K-fold ensemble improves stability of risk scoring and ranking over a single-model baseline without additional training cost.

Smart contract auditing is a knowledge-intensive engineering activity over software artifacts, where practitioners must reason about control- and data-dependence, cross-function interactions, and deployment-time risk under limited time budgets. In this context, we aim to support pre-deployment review workflows – including audit triage and release gating – by providing high-fidelity data curation, reproducible evaluation protocols, and traversal-aware graph learning models to assist expert decision-making. We do not claim graph-based models to be the only solution for smart contract vulnerability detection; instead, we study how data fidelity and traversal-aware graph modeling affect robustness under a controlled and reproducible pipeline. The main contributions are summarized as follows:

1. **Artifact-verified data curation for assurance workflows.** We automate the intake of Etherscan-verified contract artifacts and enforce solc-based compilation and structural validation to produce class-balanced datasets with traceable label provenance for four representative vulnerability categories (Section 3.3; Table 1).
2. **AOT pipeline that operationalizes scalable, reproducible screening.** We implement an ahead-of-time preprocessing workflow that materializes semantic graphs and features once offline, enabling practical K-fold evaluation and consistent re-execution of baselines under a fixed protocol (Section 3.5.1; Sections 4.1.2–4.2).
3. **Traversal-aware dual-channel representation that separates structure from ordered evidence.** We support vulnerability screening by learning (i) topology-driven dependency aggregation and (ii) traversal-derived edge-path semantics in parallel, and we provide controlled evidence on when BFS or DFS better aligns with the semantic complexity of vulnerability patterns (Section 3.5; Section 5.2.3; Tables 6–7).
4. **Zero-extra-training robustness via fold-reuse ensembling.** We provide a K-fold ensemble variant that reuses cross-validation models to stabilize inference-time risk scoring, yielding consistent gains across tasks while explicitly quantifying the inference overhead (Section 3.5.3; Tables 8–9).

**Results overview.** Under the proposed protocol on four common vulnerability types, VulGNN-SG achieves an overall F1-score of 92.69%. On the Unchecked Low Calls task, serial 5-fold training takes ∼2.5 h in total (about ∼0.5 h per fold). VulGNN-SG+ yields a +3.06% absolute F1 improvement, at the cost of roughly 5× inference latency (about ∼10 s vs. ∼2 s on the full test set).

## 2. Related work

The fundamental principles and key enabling technologies for smart contract vulnerability detection span multiple paradigms, including program-analysis-based tools (e.g., symbolic execution, static analysis, and rule-based checkers), formal verification, and learning-based approaches that model code as either sequences or structured graphs. Graph-based learning is therefore not the only viable approach; rather, it is one representative line that is particularly suitable when non-local control/data dependencies must be modeled explicitly. Accordingly, this section reviews graph-based code representations and GNNs, and positions our framework within the broader landscape.

This section presents an overview of the research progress in these areas, highlights recent advances in smart contract vulnerability detection, and discusses existing challenges in the field, with a particular emphasis on identifying limitations that motivate the research gaps addressed in this work.

### 2.1. Common smart contract vulnerabilities

The immutability of smart contracts makes thorough auditing before deployment crucial. Among various vulnerability types, several have attracted significant attention due to their high frequency and severe impact. This study focuses on four representative categories, each exhibiting distinct semantic structures and substantial influence in real-world attacks [6,7].

(1) Reentrancy Vulnerability. This occurs when a contract invokes an untrusted external contract before updating critical states (e.g., account balances). The external contract may recursively re-enter the original function, repeatedly executing withdrawal operations before the state is updated, thereby depleting funds. This vulnerability strongly depends on the control flow and the execution order between state updates and external calls.

(2) Arithmetic Vulnerability. Originating from fixed-size integers in the Ethereum Virtual Machine (EVM), this vulnerability arises when arithmetic operations exceed the upper or lower bounds of a data type, causing overflow or underflow. The resulting value wraparound allows attackers to manipulate key computational results such as balances or state variables. Detecting such vulnerabilities typically requires tracking data flow dependencies between operations.

(3) Timestamp Dependency Vulnerability. When critical logic, such as fund unlocking or outcome determination, depends on the global variable *block.timestamp*, miners can manipulate this value within allowable limits to influence execution outcomes for personal gain. Detection requires identifying control flow paths affected by timestamp dependencies.

(4) Unchecked Low-Calls Vulnerability. Low-level functions such as *call*(), *delegatecall*(), and *send*() do not throw exceptions upon failure but instead return a Boolean value `false`. If developers fail to explicitly check this return value, failed calls may be ignored, leading to transfer errors or inconsistent state updates. This vulnerability is typically reflected as a local structural flaw.

### 2.2. Graph-based code representation

Modeling source code as a graph structure is an effective representation approach that captures rich syntactic and semantic information, surpassing the expressive limitations of linear token sequences. In program analysis and vulnerability detection, several key graph representations are commonly used:

(1) **Abstract Syntax Tree (AST).** An AST represents the syntactic hierarchy of source code in a tree structure, where each node corresponds to a specific syntactic construct such as function declarations, expressions, or identifiers. While it effectively captures grammatical rules, it lacks explicit modeling of control and data flows [8].

(2) **Control Flow Graph (CFG).** A CFG illustrates the possible execution paths of a program during runtime. Nodes usually represent basic blocks, and edges denote control transfers between them, describing the execution order and logical dependencies among statements.

(3) **Data Flow Graph (DFG).** A DFG models data propagation within a program. Nodes represent variables or operations, and edges connect variable definitions to their corresponding usage points, revealing inter-statement data dependencies.

(4) **Symbolic Graph (SG).** An SG is an integrated representation that enriches an AST by introducing control-flow and data-flow edges, thereby unifying syntactic, control, and data dependencies within a single graph structure [9].

In ours *VulGNN-SG*, a function-level AST is first constructed, followed by the addition of control-flow edges to represent execution order and data-flow edges to capture variable definition–on–use relationships. Compared with single-structure graphs, the multi-relational SG provides richer semantic context for GNN learning, aligning with prior findings on the effectiveness of context-aware representations in software defect detection [10].

However, reported gains are often confounded by upstream factors (e.g., compilation consistency, graph-construction stability, and duplicate leakage across splits) that are underspecified in prior work. Moreover, traversal-derived path semantics are commonly treated as an implementation detail, leaving their impact on different vulnerability categories insufficiently characterized under a unified and reproducible protocol.

### 2.3. Applications of graph neural networks in program analysis

Graph Neural Networks (GNNs) are a class of deep learning models designed for graph-structured data [11]. Their core mechanism involves iterative message passing and state updating to learn node representations, thereby capturing high-order dependencies and semantic information embedded in graph topologies. Owing to this capability, GNNs have demonstrated strong performance in tasks requiring structured representations, such as program analysis.

The Graph Network (GN) framework [12] provides a unified theoretical foundation for various GNN architectures by defining message-passing processes through configurable *aggregate* and *update* functions. Under this framework, several representative architectures have emerged. For instance, GraphSAGE [13] introduces an inductive learning paradigm that generates node embeddings by sampling and aggregating neighborhood features, enabling efficient inference over large-scale dynamic graphs.

Common foundational GNN models include:

(1) Graph Convolutional Network (GCN). GCNs update node embeddings by aggregating normalized features of neighboring nodes, achieving isotropic representation learning with high computational efficiency [14]. In this study, the implementation is based on the `GCNConv` module from the PyTorch Geometric library.

(2) Graph Attention Network (GAT). GATs incorporate attention mechanisms into the aggregation process, enabling adaptive weighting of different neighbors to emphasize critical dependencies [15,16].

(3) Relational Graph Convolutional Network (RGCN). RGCNs are designed for multi-relational graphs, learning distinct transformation weights for each relation type to capture diverse semantic connections [17,18].

### 2.4. Smart contract vulnerability detection

Smart contract vulnerability detection has evolved from traditional static analysis to deep learning–based intelligent analysis systems. Early tools such as Oyente [19], Slither [20], and SmartCheck [21] employed symbolic execution and rule-based pattern matching, establishing the foundation for automated vulnerability analysis. However, their effectiveness can be constrained by practical failure modes, including limited scalability on complex execution paths, incomplete rule coverage for evolving vulnerability patterns, and brittleness under heterogeneous compiler versions and language features.

With the advancement of deep learning, researchers began leveraging neural networks to automatically extract program features. One line of work models source code as sequential data, using RNNs or Transformers for semantic learning; another models code as graph structures and employs GNNs to capture structured dependencies [22]. Sequence-based models can capture lexical regularities and local idioms, yet they do not explicitly encode control- and data-dependence relations, which becomes a bottleneck for vulnerabilities that are path-sensitive or rely on non-local interactions (e.g., def–use chains and state-update ordering around external calls). Graph-based detectors alleviate this limitation by representing code with explicit relational structure; nevertheless, their empirical gains may be sensitive to dataset fidelity (e.g., non-compilable sources, inconsistent compilation settings, and duplicate samples) and to evaluation practices that inadvertently inflate generalization.

Recent studies have proposed multi-view detection frameworks that jointly learn both topological and path-based semantic features of code graphs, thereby enhancing the representation capability in both structural and semantic dimensions. Among these, dual-channel parallel learning has become a key research direction. This paradigm typically constructs two independent learning branches: One aggregates neighborhood information to capture structural dependencies, while the other extracts contextual semantics from edge sequences or traversal paths. By fusing these features at the representation level, the model can obtain complementary semantic views and develop a deeper understanding of program logic [23].

Building on prior dual-channel detectors, we present VulGNN as a reproducible reference framework for traversal-aware graph learning in smart-contract vulnerability detection. Rather than claiming a new primitive operator, we focus on (i) a compilation-verified data curation workflow with traceable labels, (ii) an ahead-of-time preprocessing pipeline that makes large-scale cross-validation practical, and (iii) a controlled analysis that treats BFS/DFS traversal as a first-class modeling factor. The goal is to provide a verifiable basis for comparing dual-channel paradigms under consistent data and implementation conditions, and to derive actionable implications on when breadth- or depth-oriented semantics are beneficial.

Although prior work has substantially improved detection accuracy, three gaps remain insufficiently addressed in a unified and reproducible setting: (1) *Dataset fidelity and traceability:* existing corpora may include non-compilable contracts, inconsistent compilation configurations, and duplicates, which can undermine graph construction and bias evaluation; (2) *Evaluation reliability:* differences in preprocessing, split policies, and leakage control make cross-paper comparisons difficult and can

overestimate generalization; (3) *Traversal-sensitive semantic modeling:* the influence of traversal-derived path semantics on different vulnerability categories is often not isolated, limiting interpretability and systematic design.

Compared with existing studies, VulGNN-SG improves model reproducibility and generalization through a high-quality data curation pipeline and standardized graph construction. By modeling structural dependencies and semantic contexts through parallel channels, and employing an organized Ahead-of-Time (AOT) preprocessing strategy and robust evaluation mechanisms, the model achieves a new balance between accuracy and computational efficiency. The goal of this work is to establish a reliable, efficient, and verifiable benchmark implementation for dual-channel vulnerability detection models, providing an extensible foundation for future studies.

**Summary and positioning.** Recent detectors differ in graph construction, feature sources, and fusion strategies. For a component-level comparison with representative baselines and recent SOTA (including Devign, GraphCodeBERT, EA-RGCN, PHSCS, and HAM), we summarize key architectural and pipeline choices in Table 5. This comparison highlights that VulGNN-SG shares the high-level dual-channel motivation with prior designs, while emphasizing compilation-verified and traceable data curation, an ahead-of-time (AOT) preprocessing protocol for reproducible cross-validation, and traversal-derived edge-path semantics as an explicit and analyzable modeling factor.

## 3. Methodology

This paper presents VulGNN, a high-fidelity graph neural network framework for smart contract vulnerability detection. Centered on high-quality data curation, semantic graph feature modeling, and a dual-channel GNN architecture, VulGNN aims to establish a reproducible, extensible, and robust paradigm for smart contract vulnerability analysis.

This section first provides a formal definition of the vulnerability detection task. It then introduces the framework's three-phase workflow:

1. **Data Curation Phase** — Constructs a high-fidelity, balanced, and reproducible smart contract dataset, establishing a reliable foundation for model training.
2. **Feature Modeling Phase** — Integrates syntactic, control-flow, and data-flow information to generate multi-level graph semantic representations, enabling the capture of global program context.
3. **Model Design Phase** — Focuses on the dual-channel architecture of the core model *VulGNN-SG* and its integrated extension *VulGNN-SG+*, achieving joint improvements in performance and robustness through structural optimization and model ensemble strategies.

These three phases collectively form the complete research loop of the VulGNN framework.

### 3.1. VulGNN as an artifact-centered assurance workflow component

To align the contribution with artifact-centered engineering informatics, we position VulGNN as a workflow component that automates and supports knowledge-intensive inspection of an executable digital artifact. In blockchain-enabled engineering systems, a smart contract is an executable artifact that encodes operational rules (e.g., asset transfer logic and protocol constraints) and is deployed under stringent immutability constraints. Consequently, pre-deployment assurance requires reviewers to (i) verify artifact executability under the intended toolchain, (ii) inspect structured dependencies that govern safety-critical actions, and (iii) produce traceable screening outcomes that support audit triage and release gating.

As shown in Fig. 1, the VulGNN pipeline is organized into five stages that map onto a generic assurance workflow: (1) Engineering Artifact Intake (Step 1); (2) Artifact Verification and Curation (Steps 2–4); (3) Artifact Representation; (4) Dual-Channel Learning and Inference; (5) Decision Support for Knowledge-Intensive Review.

Fig. 1 summarizes how the proposed pipeline operationalizes this assurance workflow using only artifact-grounded steps implemented and evaluated in this paper. The workflow begins with artifact intake from Etherscan-verified source code and associated metadata, followed by verification and curation that enforces executable logic checks, compilation via the official solc toolchain, and file-level de-duplication (Sections 3.3–3.4). Verified artifacts are then converted into an artifact representation as a multi-relational semantic graph (SG) that unifies AST, control-flow, and data-flow dependencies (Sections 3.2–3.4). On top of this representation, VulGNN performs dual-channel learning and inference: a topology-driven branch aggregates global dependency structure, while a traversal-derived branch models ordered edge-path semantics (Section 3.5). The workflow produces decision support outputs – vulnerability probabilities and a ranked triage signal – together with traceability hooks (compilation records, graph snapshots, protocol configuration) that enable reproducible screening and controlled evaluation (Sections 4.1.2–4.2; Tables 6–9).

The table embedded in Fig. 1 summarizes how each workflow stage responds to a specific failure mode identified in Section 1, together with the corresponding empirical evidence.

### 3.2. Formalization of vulnerability detection as graph classification

The smart contract vulnerability detection problem is formalized as a supervised graph classification task.

Let a smart contract $C$ be represented as an attributed, multi-relational semantic graph (SG), denoted by $G = (V, E_{\text{typed}}, X)$, where $V = \{v_1, v_2, \ldots, v_N\}$ is the set of nodes representing syntactic units in the code (e.g., expressions, statements). $E_{\text{typed}} \subseteq V \times V \times R$ is the set of typed edges, where $R$ denotes the set of relation types, including three primary categories: (1) AST edges: capture the syntactic hierarchical structure of the code. (2) Control Flow (CFG) edges: represent the sequential execution paths among program statements. (3) Data Flow (DFG) edges: model the variable *define–use* relationships.

The node feature matrix $X \in \mathbb{R}^{N \times d}$ consists of $d$-dimensional feature vectors, where each row $x_i$ corresponds to node $v_i$.

For a given vulnerability type $\tau$, each graph $G$ is assigned a binary label $y \in \{0, 1\}$. The objective is to learn a GNN model $f_\theta$ that takes the graph $G$ as input and predicts its label

$$\hat{y} = f_\theta(G). \tag{1}$$

During training, the model parameters $\theta$ are optimized by minimizing the binary cross-entropy loss between the predicted and ground-truth labels. An arithmetic vulnerability code fragment can be represented as a Semantic graph, shown as in Fig. 2.

The syntactic-structure edges (AST, solid gray) outline the hierarchical organization of the code and indicate that all statement nodes belong to the `EthICO` contract. On top of this backbone, the control-flow path (CFG Path, dashed blue) explicitly captures the linear execution order (① → ② → ③). The data-flow path (DFG Path, dashed red) further establishes semantically meaningful shortcuts by connecting variable definition sites to their corresponding uses: the definition sites of variables $b$ and $c$ (`VariableDeclaration`) are linked to their use in the expression $a = b + c$ (paths ❶ and ❷), and the definition site of $a$ (`Assignment`) is linked to its use in `transfer(a)` (path ❸). This tri-relational representation enables a GNN to integrate static syntax with execution logic and def-use dependencies, thereby facilitating the learning of vulnerability-relevant, high-level semantic patterns from the SG.

This triadic graph representation enables the GNN model to comprehend not only the syntactic structure of the code but also its execution

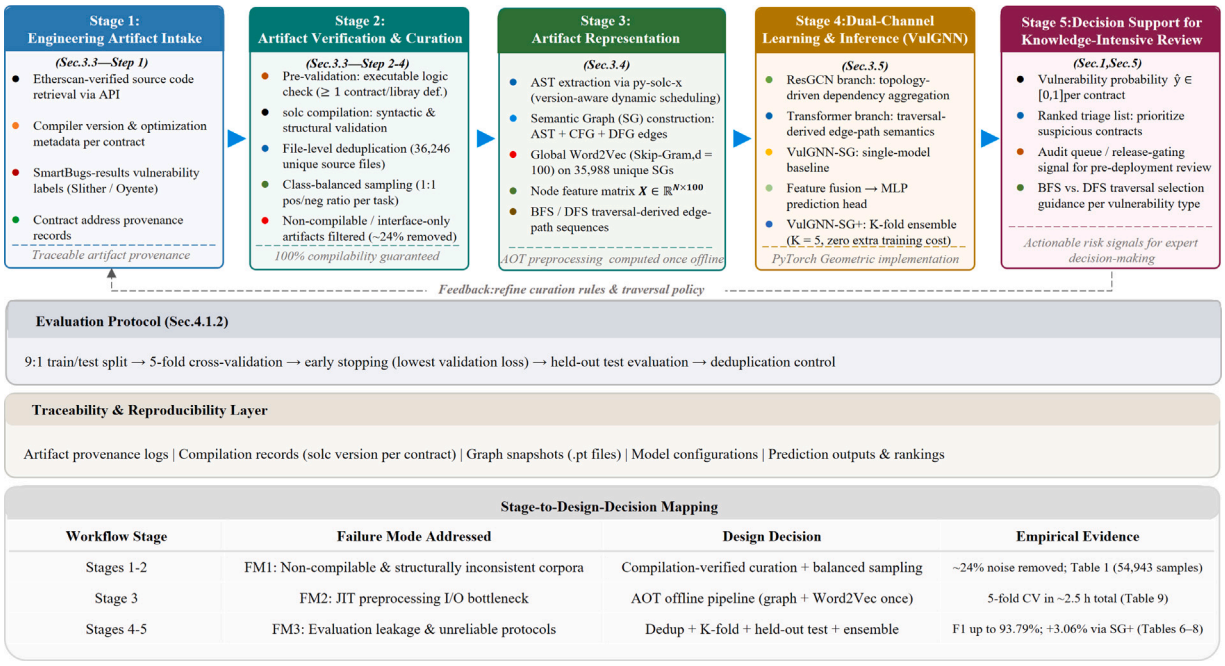



**Fig. 1.** Artifact-centered assurance workflow for VulGNN. The pipeline is organized as an engineering-informatics workflow component: artifact intake (Etherscan-verified source code), compilation-verified curation and de-duplication, semantic-graph representation (AST+CFG+DFG), traversal-aware dual-channel learning/inference (VulGNN-SG/VulGNN-SG+), and decision support outputs for audit triage and release gating under a fixed evaluation protocol.

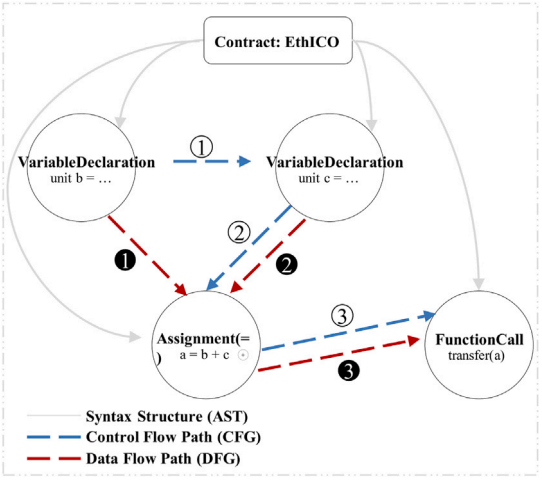



**Fig. 2.** Semantic graph representation of a representative arithmetic-related code fragment (illustrating CFG/DFG traversal traces).

logic and data dependencies. The core objective of the GNN model $f_\theta$ is to automatically learn and identify high-level semantic patterns from such enriched graph structures—for instance, detecting risky scenarios where *an unchecked arithmetic result* (node *Assignment*) directly flows, through a DFG edge, into a *value-transfer call/sensitive operation* (node *FunctionCall*).

A running example is provided in Fig. 6-Running case (EthICO) to demonstrate how SG edges encode a multi-hop dependency chain that later drives traversal-derived evidence and screening decisions.

### 3.3. Data curation: High-fidelity dataset construction pipeline

To address the pervasive issues of data quality and reliability in public smart contract repositories, this study designs and implements a systematic multi-stage data curation pipeline for constructing a high-fidelity experimental dataset. The pipeline aims to produce data resources that are structurally sound, class-balanced, and highly reproducible, ensuring the scientific validity and fairness of vulnerability detection model evaluation.

1. **Acquisition of Verified Data Sources.** The foundation of the pipeline is built upon the Etherscan API, supplemented by contract address information provided in the SmartBugs report [24]. Unlike approaches relying on static or non-compilable public code repositories, this pipeline programmatically retrieves verified source codes and their associated metadata, enabling high-trust integration of contract-level corpora. This step ensures data authenticity, semantic completeness, and strong traceability.
2. **Ground-truth labels and task definition.** The ground-truth labels used for supervised training are inherited from the *smartbugs-results* annotations associated with the SmartBugs report [22]. These annotations are generated by established analyzers (e.g., Slither and Oyente) and indicate whether a contract is flagged for a specific vulnerability category. In this work, we focus on four vulnerability types (Arithmetic, Reentrancy, Timestamp Dependency, and Un-checked Low Calls) and construct one binary classification task per type. For a target type $v$, a contract is treated as a *positive* sample if it is labeled with $v$ in *smartbugs-results*. A contract is treated as a *negative* sample for task $v$ only if it is not labeled with $v$ in *smartbugs-results* and it passes the same verification and compilation checks as positives. Importantly, the subsequent curation stages (pre-validation, compilation, and structural filtering) do not create new labels; rather, they remove unreliable instances so that the final dataset preserves traceable labels while improving data fidelity.
3. **Rigorous Pre-Verification Mechanism.** Each downloaded source file undergoes an automated pre-verification process before being included. A file is considered valid only if it: (i) contains at least one contract or library definition, excluding interface-only files without executable logic; and (ii) successfully compiles using the official `solc` compiler, passing both syntactic and structural validation. This procedure ensures high-quality

**Table 1**
Statistical summary of the final curated datasets.

| Vulnerability type | Positive | Negative | Total | Pos/Neg ratio |
|---|---|---|---|---|
| Arithmetic | 12,771 | 12,496 | 25,267 | ~1:0.98 |
| Reentrancy | 8483 | 8459 | 16,942 | ~1:1 |
| Timestamp dependency | 1421 | 1405 | 2826 | ~1:0.99 |
| Unchecked Low Calls | 4966 | 4942 | 9908 | ~1:0.99 |
| **Total** | **27,641** | **27,302** | **54,943** | – |

code samples and provides structural consistency and processing stability for subsequent feature extraction and model training.

4. **Task-Oriented Balanced Sampling Strategy.** Gi-ven the highly imbalanced distribution of security vulnerabilities, a balanced sampling strategy is employed for each detection task. The process involves: (i) identifying all verified positive samples; (ii) constructing a negative sample pool comprising verified contracts not labeled with the target vulnerability; and (iii) applying random undersampling to the majority class to form a training set with a 1:1 positive-to-negative ratio. This strategy effectively mitigates class bias and enhances the reliability and comparability of experimental results.

Through the aforementioned curation pipeline, we constructed four high-quality and class-balanced smart-contract source-code datasets. The dataset statistics are summarized in Table 1, which in total comprise 54,943 validated samples spanning four vulnerability detection tasks. As shown in Table 1, the positive-to-negative ratio is consistently close to 1:0.99 across all tasks, thereby providing a controlled training setting with near-balanced supervision and reduced susceptibility to class-prior bias. The corresponding pseudocode for the dataset construction procedure is provided in Algorithm 1.

**Algorithm 1** High-Fidelity Dataset Curation

**Input:** raw contract addresses $A_{\text{raw}}$ (from the SmartBugs report [22]), vulnerability labels $L$ (from *smartbugs-results*, generated by analyzers such as Slither/Oyente)
**Output:** a set of balanced, task-specific dataset manifests $D_{\text{manifests}}$

Initialize $F_{\text{verified}} \leftarrow \emptyset$
**for each** address $\in A_{\text{raw}}$ **do**
  $C_{\text{source}} \leftarrow \text{FetchFromEtherscan}(\text{address})$
  **if** $\text{PreValidate}(C_{\text{source}})$ **then**
    $F_{\text{verified}}.\text{add}(C_{\text{source}})$
  **end if**
**end for**
Initialize $D_{\text{manifests}} \leftarrow \emptyset$
**for each** $v \in$ VulnerabilityTypes **do**
  $D_v^{\text{pos}} \leftarrow \text{SelectPositive}(F_{\text{verified}}, L, v)$
  $D_v^{\text{neg}} \leftarrow \text{SelectNegative}(F_{\text{verified}}, L, v)$
  $M_v \leftarrow \text{Balance}(D_v^{\text{pos}}, D_v^{\text{neg}})$ // 1:1 undersampling on contracts labeled as $v$ vs. contracts not labeled as $v$
  $D_{\text{manifests}}.\text{add}(M_v)$
**end for**
**return** $D_{\text{manifests}}$

This study strictly adheres to the 1:1 balanced sampling principle to eliminate potential model bias caused by class imbalance. This strategy has become a standard practice in vulnerability detection model evaluation, ensuring that performance results across different tasks remain comparable and statistically meaningful. On this basis, the study establishes performance baselines under controlled and balanced experimental conditions, providing a reliable reference for subsequent model enhancement and structural optimization.

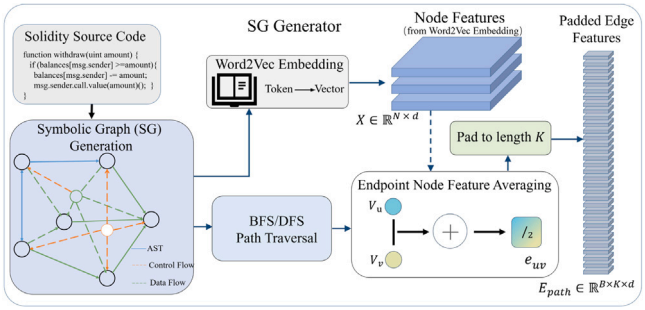



**Fig. 3.** Illustrates the pipeline from source code to graph features (SG Generator).

### 3.4. Global representation learning based on semantic graphs

The feature-engineering pipeline is a global, ahead-of-time (AOT) preprocessing stage. Its goal is to learn general, semantically rich representations from the full curated contract corpus before task-specific training. This stage provides high-level priors for subsequent model optimization and is critical to improving vulnerability detection performance and robustness. The overall pipeline is shown in Fig. 3. The workflow pseudocode is given in Algorithm 2.

**Algorithm 2** Global Graph Feature Engineering Pipeline

**Input:** dataset manifests set $D_{\text{manifests}}$
**Output:** global node embedding model $\mathcal{E}_{\text{w2v}}$, graph data file set $\mathcal{F}_{\text{graph}}$
$\mathcal{F}_{\text{unique}} \leftarrow \text{GetUniqueFiles}(D_{\text{manifests}})$ // consolidate unique sources
$\mathcal{A}_{\text{AST}} \leftarrow \text{GenerateASTs}(\mathcal{F}_{\text{unique}})$ // parse via py-solc-x
$\mathcal{G}_{\text{SG}}, \mathcal{C}_{\text{series}} \leftarrow \emptyset, \emptyset$
**for each** $ast \in \mathcal{A}_{\text{AST}}$ **do**
  $G_{\text{sg}} \leftarrow \text{BuildSemanticGraph}(ast)$
  $\mathcal{G}_{\text{SG}}.\text{add}(G_{\text{sg}})$
  $series \leftarrow \text{ExtractTokenSequence}(G_{\text{sg}})$
  $\mathcal{C}_{\text{series}}.\text{add}(series)$
**end for**
$\mathcal{E}_{\text{w2v}} \leftarrow \text{TrainWord2Vec}(\mathcal{C}_{\text{series}})$ // train on global corpus
$\mathcal{F}_{\text{graph}} \leftarrow \emptyset$
**for each** $G_{\text{sg}} \in \mathcal{G}_{\text{SG}}$ **do**
  $\mathbf{X} \leftarrow \text{GenerateNodeFeatures}(G_{\text{sg}}, \mathcal{E}_{\text{w2v}})$
  $(\mathbf{E}_{\text{bfs}}, \mathbf{E}_{\text{dfs}}) \leftarrow \text{GenerateEdgeFeatures}(G_{\text{sg}}, \mathbf{X})$
  $files \leftarrow \text{SaveGraphAndFeatures}(G_{\text{sg}}, \mathbf{X}, \mathbf{E}_{\text{bfs}}, \mathbf{E}_{\text{dfs}})$
  $\mathcal{F}_{\text{graph}}.\text{add}(files)$
**end for**
**return** $\mathcal{E}_{\text{w2v}}, \mathcal{F}_{\text{graph}}$

*From source code to Semantic Graph (SG).* In the feature construction stage, all curated, unique contract sources are parsed into abstract syntax trees (ASTs). Our SourceCode2AST tool, built on py-solc-x, automatically invokes the solc compiler version indicated by each contract's metadata. This mechanism supports dynamic version scheduling and backward-compatible fallbacks, maintaining a high parsing success rate in multi-version settings. Next, each AST is expanded into a semantic graph (SG). Unlike an AST that encodes only syntactic hierarchy, the SG augments syntax nodes with control-flow and data-flow edges to capture execution paths and variable dependencies [25]. This structured enhancement strengthens the model's ability to capture program semantics and logical relations, and provides a unified basis for subsequent graph representation learning.

*Global corpus construction and validation.* To support representation learning, we first aggregate the sample lists from the four detection tasks, resulting in 54,943 instances. After file-level de-duplication, we

obtain 36,246 unique smart contracts. To quantify the quality gains from our curation process, we conduct an overlap analysis between this unique file set and the original *smartbugs-wild* repository, which contains 47,398 files. The results indicate that our dataset covers approximately 76.16% (36,100) of the files in the original repository. This suggests that about 24% of the raw samples are automatically removed by our stringent pre-validation procedure (e.g., compilation failures, compiler-version incompatibilities, or the absence of executable logic). This ratio not only provides a quantitative estimate of the noise level in public datasets but also corroborates the structural integrity advantage of the high-fidelity subset curated in this study. We then submit all 36,246 validated source files to the semantic graph (SG) generation pipeline, retaining only graphs that are successfully constructed and satisfy the non-triviality constraint (number of entry nodes >1). After this final screening step, we obtain 35,988 high-quality and unique semantic graphs, which constitute the global corpus for feature learning in this work.

*Global node and edge feature embedding.* Using the global corpus of 35,988 semantic graphs (SGs) as the training base, we adopt a fine-grained tokenization scheme that encodes each node by combining its type and identifier name, yielding a vocabulary of 519,119 unique tokens. We then train a global Word2Vec (Skip-Gram) model to learn a 100-dimensional vector for each token. Compared with a coarse-grained, type-only scheme, this fine-grained representation captures semantic distinctions more effectively and is a key factor influencing model performance [26]. The learned embeddings form the graph-level node feature matrix, where $N$ is the number of nodes and the embedding dimension is $d = 100$.

We also construct initial edge features. For an edge $(u, v)$ with node features $\mathbf{x}_u, \mathbf{x}_v \in \mathbb{R}^d$, the initial edge feature is

$$\mathbf{e}_{uv} = \frac{1}{2}(\mathbf{x}_u + \mathbf{x}_v), \tag{2}$$

Node and edge features are jointly used as inputs to the VulGNN-SG encoder.

### 3.5. VulGNN: High- fidelity graph neural network framework

The framework comprises two core modules: (1) VulGNN-SG, a dual-channel encoder that integrates graph representation learning with edge-feature extraction; and (2) VulGNN-SG+, an ensemble architecture with five-fold aggregation for robustness.

#### 3.5.1. Implementation and protocol enhancements in VulGNN-SG: Layer-wise feature concatenation and an AOT pipeline

VulGNN-SG is implemented in PyTorch Geometric (PyG) and serves as the base model in our high-fidelity VulGNN framework. This component focuses on methodological and reproducibility aspects of semantic-graph learning rather than proposing new message-passing operators. We therefore detail two implementation/protocol choices that (i) stabilize training on semantic graphs and (ii) make large-scale evaluation reproducible.

**(1) Architectural choice: layer-wise feature concatenation in ResGCN.** To alleviate oversmoothing in deeper GCN stacks and to retain multi-scale structural information, the ResGCN branch adopts a layer-wise concatenation strategy: we concatenate the output feature maps from all graph convolution layers to form the final node representation. Compared with residual designs based on element-wise summation, concatenation preserves features from different receptive-field depths explicitly, allowing the classifier to access both local patterns and longer-range structural cues.

**(2) Evaluation-enabling choice: an ahead-of-time (AOT) preprocessing pipeline.** Graph-based vulnerability detection often suffers from substantial just-in-time (JIT) overhead due to repeated parsing and graph construction during training. We therefore adopt an ahead-of-time (AOT) workflow in which compute-intensive preprocessing steps – including AST parsing, semantic graph construction, and Word2Vec feature generation – are performed once offline. Training then loads preprocessed, GPU-friendly data objects, decoupling learning from preprocessing and enabling scalable $K$-fold cross-validation under a fixed and reproducible protocol.

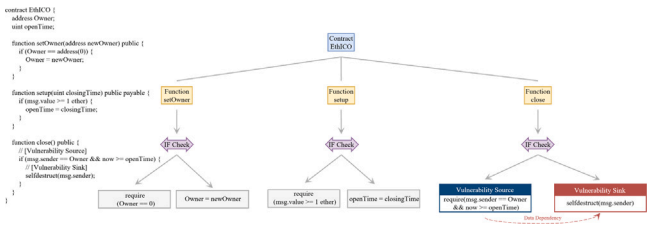



**Fig. 4.** A motivating example of path-compositional vulnerability semantics: a time-related predicate participates in a multi-hop dependency chain that gates a sensitive operation (`selfdestruct`).

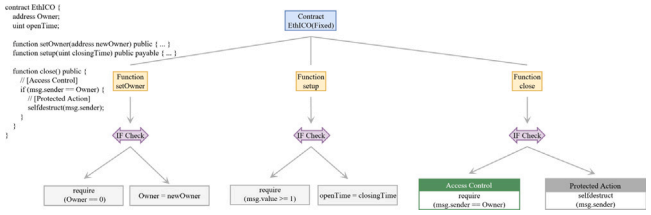



**Fig. 5.** Patched logic after dependency removal: the security decision becomes an explicit access-control guard directly protecting the sensitive action, shortening the dependency chain.

**(3) Dual-channel modeling on high-fidelity semantic graphs: a motivating dependency-chain example.** In addition to the above choices, VulGNN-SG employs a dual-channel encoder that combines a topology-oriented ResGCN branch with an edge-semantics Transformer branch. These two code-grounded examples also illustrate why vulnerability evidence can be either local (favoring BFS) or path-compositional (favoring DFS), which motivates the traversal analysis in Section 5.2.3. Figs. 4 and 5 provide a concrete example (EthICO) that motivates why modeling *path-compositional semantics* is important. In the vulnerable version (Fig. 4), a time-related predicate (`now`) participates in a multi-hop dependency chain that gates a sensitive operation (`selfdestruct`). Such vulnerability evidence is naturally expressed as a source-to-sink dependency path traversing intermediate control checks and state updates, rather than as an isolated node attribute. In the patched version (Fig. 5), this dependency chain is removed and replaced by an explicit access-control guard, shortening the semantic path and eliminating the original source-to-sink dependency. This example serves as mechanistic support for our traversal-aware design and the traversal analysis reported in Section 5.2.3.

For continuity, a running case (EthICO) shown in Fig. 6 traces the same artifact through intake/verification, SG construction, traversal-derived sequencing, and the final triage-oriented output under the fixed evaluation protocol.

As shown in Fig. 6, a running case (EthICO): from verified artifact to triage decision. We illustrate a cohesive end-to-end narrative using the EthICO timestamp-dependency example (Figs. 4–5). The artifact enters the pipeline as Etherscan-verified source code with vulnerability labels inherited from smartbugs-results (Section 3.3). During intake and verification, the artifact is retained only if it contains executable logic and compiles under the official solc toolchain, ensuring that downstream parsing and SG construction are stable (Sections 3.3–3.4). The verified artifact is then represented as an SG that explicitly encodes the multi-hop dependency chain from a time-related predicate (e.g., now/block.timestamp) through intermediate checks to a sensitive

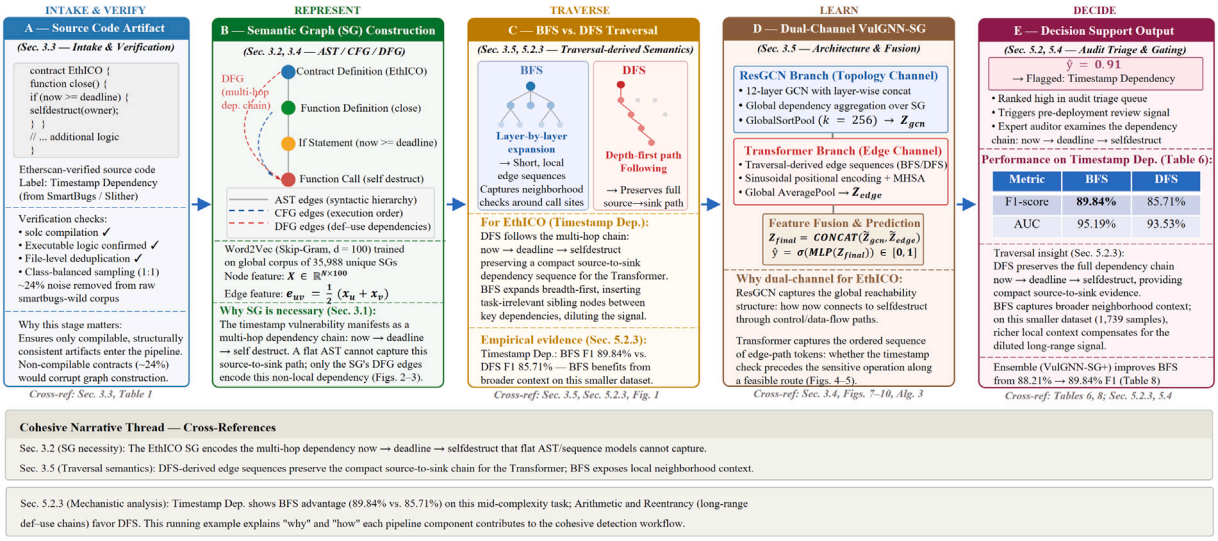



**Fig. 6.** Running case study: EthICO contract through the VulGNN pipeline.

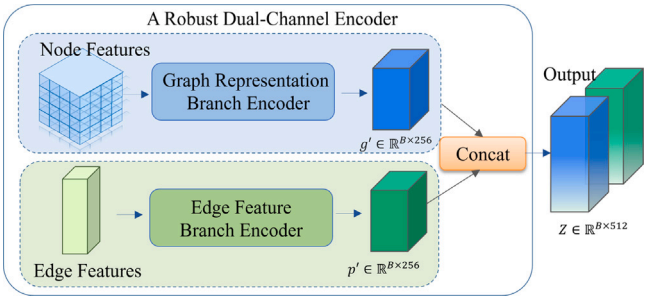



**Fig. 7.** The overall architecture of the VulGNN.

operation (e.g., selfdestruct), which cannot be reliably characterized by a flat token sequence alone (Section 3.2; Figs. 4–5). On this SG, the topology channel aggregates global reachability and dependency structure, while the traversal-derived channel produces ordered edge-path sequences for the Transformer. The BFS/DFS choice changes which evidence is made salient: DFS preserves compact source-to-sink chains when vulnerability semantics are path-compositional, whereas BFS exposes richer local neighborhoods around critical call sites (Section 3.5; Section 5.2.3). The resulting model output is a vulnerability probability that can be used as a triage signal for pre-deployment review under a fixed protocol (Sections 4.1.2–4.2), and the controlled experiments quantify how this traversal-induced evidence interacts with vulnerability categories (Tables 6–8).

### 3.5.2. *VulGNN- SG*

VulGNN-SG is implemented with PyG and serves as the base model of the framework. Its overall architecture is shown in Fig. 7. The core forward pass is summarized in Algorithm 3.

As illustrated in Fig. 7, VulGNN-SG adopts a dual-channel encoding design with a *Graph Representation Branch* and an *Edge Feature Branch*. The rationale is that vulnerability evidence typically arises from two complementary sources: (i) global dependency structure (e.g., control/data-flow connectivity that links definitions, checks, and critical calls), and (ii) ordered semantic cues that become salient along execution-related paths (e.g., whether a check precedes a sensitive operation along a feasible route). A monolithic encoder that fuses heterogeneous signals early may entangle these cues and obscure which dependencies drive predictions. Accordingly, the Graph Representation Branch assigns topology-driven aggregation to ResGCN, whereas the

**Algorithm 3** VulGNN-SG Forward Propagation

**Input:** batch of graphs with node features $\mathbf{X}$, adjacency $\mathbf{A}$, and edge sequence $\mathbf{E}$
**Output:** batch of prediction scores $\hat{\mathbf{Y}}$
*// Graph Representation Branch*
$\mathbf{H}_{\text{res}} \leftarrow \text{ResGCN}(\mathbf{X}, \mathbf{A})$ // Eq. (2)–(4)
$\mathbf{z}_{\text{gcn}} \leftarrow \text{GlobalSortPool}(\mathbf{H}_{\text{res}}, k)$
*// Edge Feature Branch*
$\mathbf{H}_{\text{edge}} \leftarrow \text{EdgeAttention}(\mathbf{E})$ // Eq. (5)–(7)
$\mathbf{z}_{\text{edge}} \leftarrow \text{GlobalAveragePool}(\mathbf{H}_{\text{edge}})$
*// Fusion and Prediction Head*
$\tilde{\mathbf{z}}_{\text{gcn}} \leftarrow \text{ReLU}\big(\text{Linear}_{\text{gcn}}(\text{Flatten}(\mathbf{z}_{\text{gcn}}))\big)$
$\tilde{\mathbf{z}}_{\text{edge}} \leftarrow \text{ReLU}\big(\text{Linear}_{\text{edge}}(\mathbf{z}_{\text{edge}})\big)$
$\mathbf{z}_{\text{final}} \leftarrow \text{CONCAT}(\tilde{\mathbf{z}}_{\text{gcn}}, \tilde{\mathbf{z}}_{\text{edge}})$
$\hat{\mathbf{Y}} \leftarrow \sigma\big(\text{MLP}(\mathbf{z}_{\text{final}})\big)$
**return** $\hat{\mathbf{Y}}$

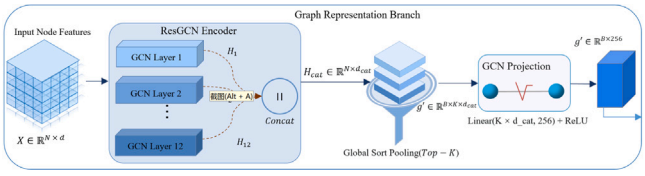



**Fig. 8.** Schematic of the dual-channel graph representation branch.

Edge Feature Branch uses Transformer-style self-attention to model order-sensitive context along traversal-derived sequences. The complementarity of the two branches is validated by the ablation study in Section 4, where removing either branch leads to a substantial degradation in performance.

*Graph representation branch (ResGCN).* This branch learns high-level node semantics from graph topology. As shown in Fig. 8, it stacks $L$ graph convolution layers (GCNs). For the $l$th layer ($l \in \{1, 2, \ldots, L\}$), the node feature matrix $\mathbf{H}^{(l)} \in \mathbb{R}^{N\times d_l}$ is updated by the standard GCN propagation:

$$\mathbf{H}^{(l)} = \hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l-1)}\mathbf{W}^{(l)}, \tag{3}$$

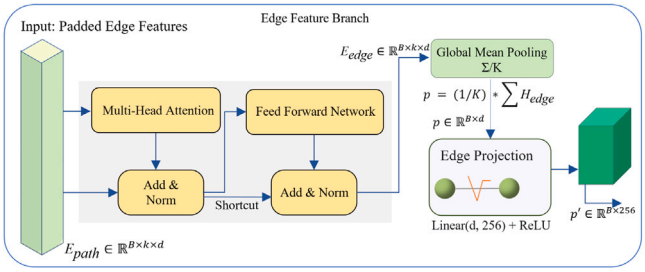



**Fig. 9.** Schematic of the dual-channel edge feature branch.

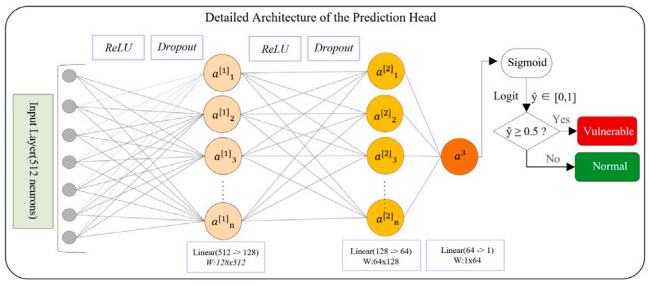



**Fig. 10.** Detailed architecture of the prediction head.

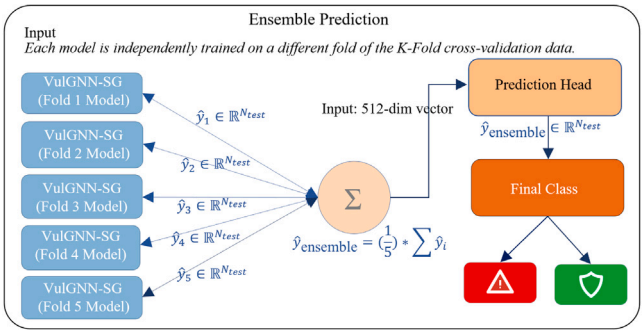



**Fig. 11.** Schematic of the ensemble prediction framework.

where $\mathbf{H}^{(l-1)}$ is the previous-layer feature matrix ($\mathbf{H}^{(0)} = \mathbf{X}$); $\mathbf{W}^{(l)}$ is a learnable weight matrix; $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency with self-loops; and $\hat{\mathbf{D}}$ is the corresponding degree matrix. To enhance nonlinearity, all layers except the output layer ($l < L$) apply the tanh activation:

$$\mathbf{H}'^{(l)} = \tanh\left(\mathbf{H}^{(l)}\right). \tag{4}$$

This design is motivated by the observation that vulnerability-relevant dependencies may reside at different receptive-field scales, while deep message passing can suffer from over-smoothing on large semantic graphs. To mitigate over-smoothing and capture multi-scale structure, we concatenate layer outputs along the feature dimension to form a residual representation:

$$\mathbf{H}_{\text{res}} = \text{Concat}\left(\mathbf{H}'^{(1)}, \mathbf{H}'^{(2)}, \ldots, \mathbf{H}'^{(L-1)}, \mathbf{H}'^{(L)}\right), \tag{5}$$

where $\mathbf{H}_{\text{res}} \in \mathbb{R}^{N \times d_{\text{total}}}$. Finally, $\mathbf{H}_{\text{res}}$ is fed to Global Sort Pooling [27], which selects the top-$k$ nodes by sorted features to produce a fixed-size graph representation $\mathbf{Z}_{\text{gcn}} \in \mathbb{R}^{k \times d_{\text{total}}}$.

*Edge feature branch (Edge_Attention).* The edge feature branch is shown in Fig. 9. This parallel branch models contextual semantics from traversal-derived edge sequences. The input sequence consists of edge features produced by BFS or DFS, denoted $\mathbf{E} \in \mathbb{R}^{k_{\text{edge}} \times d_{\text{feat}}}$, where $k_{\text{edge}}$ is the padded length. BFS prioritizes neighborhood-level ordering and tends to expose local semantic patterns around sensitive primitives (e.g., whether a low-level call is immediately followed by a return-value check). DFS unfolds longer execution-related paths and can surface non-local dependency evidence (e.g., def–use chains or state-update ordering that spans multiple control-flow steps) [28]. By converting graph edges into an ordered sequence, this branch enables self-attention to model relative order and contextual co-occurrence that are not directly represented by permutation-invariant neighborhood aggregation.

To encode order explicitly, we adopt sinusoidal positional encoding (PE):

$$\begin{aligned} \text{PE}(\text{pos}, 2i) &= \sin!\left(\text{pos}/10000^{\cdot 2i/d\text{feat}}\right) \\ \text{PE}(\text{pos}, 2i+1) &= \cos!\left(\text{pos}/10000^{\cdot 2i/d\text{feat}}\right), \end{aligned} \tag{6}$$

The position-enhanced sequence $\mathbf{E}' = \mathbf{E} + \text{PE}$ is fed into a Transformer encoder, where multi-head self-attention (MHSA) captures higher-order dependencies among edges:

$$\mathbf{AttnOut} = \text{MHSA}(\text{LayerNorm}(\mathbf{E}')) + \mathbf{E}'. \tag{7}$$

A feed-forward network (FFN) with residual connection then yields the final edge representation:

$$\mathbf{H}_{\text{edge}} = \text{FFN}(\text{LayerNorm}(\mathbf{AttnOut})) + \mathbf{AttnOut}. \tag{8}$$

Finally, $\mathbf{H}_{\text{res}}$ is fed to Global Sort Pooling [27], which selects the top-$k$ nodes by sorted features to produce a fixed-size graph representation $\mathbf{Z}_{\text{gcn}} \in \mathbb{R}^{k \times d_{\text{total}}}$. This operation yields a length-invariant summary while retaining salient node responses, which is necessary because contract/function graphs vary substantially in size.

### 3.5.3. VulGNN-SG+

To connect the mechanistic analysis to a concrete artifact, we refer back to Fig. 6 (EthICO), where the timestamp-related dependency chain exemplifies how traversal order changes the compactness of source-to-sink evidence made available to the edge-semantic channel. The dual-channel design yields two graph-level embeddings, $\mathbf{z}_{\text{gcn}}$ and $\mathbf{z}_{\text{edge}}$. Each is first projected to a shared 256-dimensional space via independent linear layers, then concatenated along the feature axis to form the fused vector $\mathbf{z}_{\text{final}}$:

$$\begin{aligned} \tilde{\mathbf{z}}_{\text{gcn}} &= \text{ReLU}\left(\text{Linear}_{\text{gcn}}(\text{Flatten}(\mathbf{z}_{\text{gcn}}))\right), \\ \tilde{\mathbf{z}}_{\text{edge}} &= \text{ReLU}\left(\text{Linear}_{\text{edge}}(\mathbf{z}_{\text{edge}})\right), \\ \mathbf{z}_{\text{final}} &= \text{CONCAT}(\tilde{\mathbf{z}}_{\text{gcn}}, \tilde{\mathbf{z}}_{\text{edge}}). \end{aligned} \tag{9}$$

The fused vector is fed to a multilayer perceptron (MLP) with ReLU activations and dropout, see Fig. 10 for the architecture of the confusion and prediction head. The final vulnerability probability $\hat{y}$ is obtained by a sigmoid:

$$\hat{y} = \sigma\left(\text{MLP}(\mathbf{z}_{\text{final}})\right). \tag{10}$$

During training, we minimize the binary cross-entropy between the predicted probability $\hat{y}$ and the ground-truth label $y$:

$$\mathcal{L} = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right]. \tag{11}$$

To further improve the baseline VulGNN-SG and reduce prediction variance, we adopt an ensemble scheme, VulGNN-SG+, as illustrated in Fig. 11. It performs $K$-fold cross-validation ($K = 5$) to train $K$ independent VulGNN-SG models on different folds. For any test sample, we obtain $K$ probabilities $\{\hat{y}_i\}_{i=1}^{K}$ and compute the ensemble prediction by arithmetic averaging:

$$\hat{y}_{\text{ensemble}} = \frac{1}{K}\sum_{i=1}^{K} \hat{y}_i. \tag{12}$$

The final class label is determined by comparing $\hat{y}_{\text{ensemble}}$ with the decision threshold 0.5. This ensemble smooths idiosyncratic errors and biases of individual models, yielding a more stable and reliable prediction [29]. The corresponding pseudocode is provided in Algorithms 4.

**Algorithm 4** K-Fold Model Ensemble Prediction

**Input:** trained model set $\mathcal{M}_{\text{trained}} = \{\theta_1^*, \ldots, \theta_K^*\}$; test dataset $D_{\text{test}}$
**Output:** final ensemble prediction scores $\hat{\mathbf{Y}}_{\text{ensemble}}$
$\hat{\mathcal{Y}}_{\text{folds}} \leftarrow \emptyset$ // container for per-model predictions
**for each** $\theta_k^* \in \mathcal{M}_{\text{trained}}$ **do**
  $\hat{\mathbf{y}}_k \leftarrow \text{Predict}(\theta_k^*, D_{\text{test}})$ // probabilities from model $k$
  $\hat{\mathcal{Y}}_{\text{folds}}.\text{add}(\hat{\mathbf{y}}_k)$
**end for**
$\hat{\mathbf{Y}}_{\text{ensemble}} \leftarrow \text{Average}(\hat{\mathcal{Y}}_{\text{folds}})$ // average across $K$ models
**return** $\hat{\mathbf{Y}}_{\text{ensemble}}$

## 4. Experimental description

In smart contract vulnerability detection, traversal strategies over contract control-flow graphs fall into two categories: breadth-first search (BFS) and depth-first search (DFS). BFS excels at capturing local structural features but is limited in modeling global dependencies. In contrast, DFS models deep control-flow dependencies and is better suited to vulnerabilities distributed along cross-function call chains and execution paths. Integrating both strategies yields structural complementarity, enabling concurrent learning of local semantic features and global execution semantics. We propose VulGNN-SG, a hybrid BFS/DFS model that implements a dual-learning mechanism combining local neighborhood aggregation with deep path dependency modeling.

This chapter presents the experimental design and results to address the objectives stated in Section 1:

- RQ1: What is the detection performance of the VulGNN-SG baseline on public benchmark datasets?
- RQ2: To what extent does the ensemble variant, VulGNN-SG+, improve performance over the single-model baseline?
- RQ3: Within the dual-channel architecture, how do BFS and DFS differ in effectiveness across vulnerability types?

### 4.1. Experimental setup

#### 4.1.1. Datasets

Experiments use four class-balanced datasets constructed in Section 3.3. The vulnerability labels are inherited from *smartbugs-results* and fixed before data splitting. All subsequent filtering steps (e.g., compilation and structural consistency checks) remove invalid instances but do not modify labels. The datasets are derived from verified smart contracts on Etherscan[2] and cover four representative vulnerability categories. During conversion to the PyTorch Geometric format, we applied strict filtering. Samples were discarded if the number of graph nodes was inconsistent with the feature vector dimensionality, or if any parsing anomaly was detected. This quality control retained only structurally complete and internally consistent samples.

After converting source code into graph-structured representations using the algorithms described in Section 3.3, we conducted a statistical analysis of graph complexity. As summarized in Table 2, real-world smart-contract graphs exhibit substantial structural complexity with a pronounced long-tail distribution. The average number of nodes in each task is far larger than that of conventional GNN benchmark datasets. More importantly, we observed extreme outliers in the *Arithmetic* task: the largest graph contains more than 1.18 million nodes and 1.47 million edges.

Given the prohibitive computational cost of processing such ultra-large graphs, we applied a complexity-based filtering policy during the conversion to the PyTorch Geometric format. Specifically, we discard samples with more than 40,000 nodes or 50,000 edges. In addition, we remove samples with inconsistent graph sizes (i.e., a mismatch between the number of nodes and the feature matrix) or any parsing/serialization errors. These quality-control steps ensure computational feasibility and structural consistency of the experimental data. After filtering and dataset splitting, the final statistics are reported in Table 3. All experiments are conducted on the four curated, high-fidelity, class-balanced datasets constructed in Section 3.3.

To quantify the extent of the curation, we further measured the overlap between our curated datasets and the original SMARTBUGS-WILD repository. The curated datasets retain 36,100 unique contract files, corresponding to a coverage of 76.16% (36,100/47,398). Hence, 23.84% of the original samples are excluded due to failing strict compilation checks (e.g., syntax errors or missing version dependencies) or containing no executable logic (e.g., interface-only contracts). This exclusion rate provides a quantitative indication of the noise inherent in public code repositories. In contrast, our curated datasets guarantee 100% compilability and structural integrity, reducing the risk that invalid samples confound model training and providing a more rigorous basis for baseline evaluation.

The final statistics are reported in Table 3. All experiments are conducted on the four high-quality, high-fidelity, class-balanced datasets introduced in Section 3.3.

#### 4.1.2. Evaluation protocol

To assess generalization accurately, we adopt a rigorous protocol that combines K-fold cross-validation and a held-out test set. (1) Data split: For each vulnerability detection task, the dataset is divided at a 9:1 ratio. The 10% portion serves as an independent test set and remains isolated during training. (2) Cross-validation: The remaining 90% is evaluated with 5-fold cross-validation. We partition it into five disjoint subsets and, in turn, use four folds for training and one fold for validation, repeating this process five times so that each fold acts as the validation set once. (3) Model selection: We apply early stopping and retain, for each fold, the checkpoint with the lowest validation loss. (4) Performance assessment: The five models obtained from the 5 folds are evaluated on the held-out test set to estimate performance on unseen data.

#### 4.1.3. Experimental environment

All experiments are conducted on a server equipped with an NVIDIA RTX 4090 (24 GB) GPU, a 12-core Intel Xeon Platinum 8255C CPU, and 128 GB RAM. The software stack is based on PyTorch 1.12.1 and PyTorch Geometric 2.2.0.

#### 4.1.4. VulGNN-SG+ hyperparameter settings

The hyperparameters of VulGNN-SG+ and their roles are summarized in Table 4.

### 4.2. Baseline protocol and fair comparison

A key concern in smart-contract vulnerability detection is that many prior baselines were originally evaluated on different benchmarks and under heterogeneous preprocessing pipelines. To ensure a fair and reproducible comparison, we *re-evaluate all executable competing methods under a unified protocol*.

**Unified input representation.** All learning-based baselines are trained and tested on the same graph-structured inputs generated by our data curation and semantic graph construction pipeline (Section 3.3). This avoids confounding factors from benchmark-specific graph definitions and ensures that performance differences primarily reflect modeling choices rather than inconsistent preprocessing.

**Unified splits and label provenance.** We use identical train/validation/test splits for all methods (Table 3). Vulnerability labels are inherited from *smartbugs-results* and fixed *before* data splitting; subsequent filtering removes invalid instances (e.g., non-compilable or

[2] Etherscan: Ethereum block explorer and analytics platform. URL: https://etherscan.io.

**Table 2**
Graph-structure complexity statistics of the constructed datasets.

| Vulnerability | Category | Avg. nodes | Avg. edges | Avg. degree | Max nodes | Max edges |
|---|---|---|---|---|---|---|
| Reentrancy | Vulnerable | 17 090.78 | 21 123.89 | 1.24 | 773 345 | 961 400 |
| Reentrancy | Normal | 6 435.86 | 7 900.98 | 1.23 | 920 123 | 1 136 065 |
| Arithmetic | Vulnerable | 8 039.06 | 9 918.69 | 1.23 | 682 123 | 845 250 |
| Arithmetic | Normal | 10 813.61 | 13 243.79 | 1.22 | 1 180 369 | 1 472 744 |
| timestamp_dependency | Vulnerable | 14 535.26 | 18 043.31 | 1.24 | 234 276 | 293 750 |
| timestamp_dependency | Normal | 7 612.56 | 9 313.95 | 1.22 | 477 001 | 587 505 |
| unchecked_low_calls | Vulnerable | 8 189.50 | 10 136.55 | 1.24 | 773 345 | 961 400 |
| unchecked_low_calls | Normal | 7 885.51 | 9 685.46 | 1.23 | 966 641 | 1 185 760 |

**Table 3**
Final dataset splits used in experiments.

| Vulnerability | Split | Positive | Negative | Total |
|---|---|---|---|---|
| Arithmetic | Total | 9206 | 8850 | 18 056 |
| | Test set | 922 | 883 | 1 805 |
| | Training set | 5308 | 5092 | 10 400 |
| | Validation set | 1327 | 1273 | 2600 |
| Reentrancy | Total | 6657 | 7379 | 14 036 |
| | Test set | 689 | 748 | 1 437 |
| | Training set | 4775 | 5304 | 10 079 |
| | Validation set | 1193 | 1327 | 2 520 |
| Timestamp dependency | Total | 892 | 847 | 1 739 |
| | Test set | 90 | 85 | 175 |
| | Training set | 642 | 609 | 1 251 |
| | Validation set | 160 | 153 | 313 |
| Unchecked Low Calls | Total | 4731 | 4749 | 9 480 |
| | Test set | 477 | 471 | 948 |
| | Training set | 3403 | 3422 | 6 825 |
| | Validation set | 851 | 856 | 1 707 |

**Table 4**
Hyperparameter configuration for VulGNN-SG+.

| Hyperparameter | Description | Value |
|---|---|---|
| $N_{\text{hid}}$ | Hidden dimension of GCN layers | 24 |
| $N_{\text{rg}}$ | Number of layers in ResGCN | 12 |
| $k_{\text{sortpool}}$ | Nodes retained by SortPooling | 256 |
| `batch_size` | Graphs per training batch | 16 |
| `weight_decay` | L2 regularization strength | $1 \times 10^{-5}$ |
| $lr_{\max}$ | Maximum learning rate for CyclicLR | 0.001 |

structurally inconsistent samples) without altering labels. This guarantees that all methods see the same supervision signal under the same split indices.

**Unified training and evaluation protocol.** All baselines are trained with the same early-stopping criterion on the validation set and evaluated on the held-out test set. We report Accuracy, Precision, Recall, F1-score, and AUC (Tables 6–7). In particular, AUC is computed from continuous prediction scores (e.g., post-sigmoid probabilities) to reflect ranking-based discriminability under potential class-imbalance shifts.

**Baseline implementations and hyperparameters.** For baselines with publicly available implementations, we run the official code whenever possible; otherwise, we implement the methods following their original papers and verify correctness against the reported behavior. To prevent unfair advantages from extensive per-baseline tuning, we adopt a consistent hyperparameter search budget and select hyperparameters based on validation performance under the same search space and stopping rule. All configuration files and scripts for baseline training/evaluation are provided to facilitate reproduction.

**Handling non-executable baselines and missing metrics.** Some prior works do not release executable implementations or do not expose continuous scores required to compute ROC/AUC. In such cases, we mark the corresponding metrics as "–" in Tables 6–7 and do *not* mix incomparable numbers into the unified evaluation. We emphasize that AUC is reported for all baselines that can be executed end-to-end under our unified pipeline and that provide score outputs.

## 5. Experimental results and analysis

To validate the advantage of the VulGNN framework over existing detectors, we conduct ten independent runs for each task. We evaluate models across multiple vulnerability types and graph traversal strategies. The results reveal three levels of insights into applying GNNs to smart contract vulnerability detection.

### 5.1. Experimental setup

#### 5.1.1. Metrics

To provide a more comprehensive and fine-grained evaluation, we report five metrics: Accuracy (Acc), Precision (Pre), Recall (Rec), F1-score (F1), and the area under the receiver operating characteristic curve (AUC). We report F1-Score as the primary metric. It is the harmonic mean of Precision and Recall, reflecting both accuracy and coverage:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

#### 5.1.2. Vulnerability types

We consider four representative vulnerabilities: (1) Reentrancy, (2) Timestamp Dependency, (3) Arithmetic, and (4) Unchecked Low Calls.

#### 5.1.3. Baselines and grouping

We compare against the following detectors: PHSCS [30], HAM [31], EA-RGCN (BFS/DFS) [21], SliSE (program slicing) [32], SC-Defender (recursive traversal) [33], VSCL (explicit DFS) [34], CGE [35], DR-GCN [36], and GA-based Profiling [37]. To analyze information propagation paradigms, especially the contrast between DFS and BFS, we group the learning-based baselines into three classes by their core mechanism: DFS-based, BFS-based (layerwise propagation), and non-explicit-traversal detectors.

This grouping is introduced solely to facilitate a traversal-effect analysis under a unified learning-based evaluation protocol; it is not intended as a comprehensive taxonomy of all smart-contract detectors, which also includes traditional analyzers (e.g., symbolic execution and rule checkers) and sequence-based models whose outputs and evaluation settings are often not directly comparable in our controlled benchmark.

### 5.2. Results

#### 5.2.1. Positioning analysis.

After converting source code into graph-structured representations, recent SOTA detectors emphasize different aspects of graph construction, feature sources, and information fusion, as summarized in Table 5. For example, PHSCS and HAM leverage heterogeneous relations or sequence modeling to characterize code entities and their interactions; however, they do not explicitly model execution-path dependencies or the complementary interplay between sequential semantics and graph topology that is often critical to vulnerability triggering. GraphCodeBERT benefits from large-scale pretraining and rich semantic representations, yet its backbone is a Transformer, where the utilization of

**Table 5**
Extended comparison of representative smart-contract vulnerability detectors in terms of architectural components and pipeline design. "Explicit edge semantics" indicates whether the model encodes traversal-derived edge sequences with an order-aware module (e.g., positional encoding + Transformer). "ResGCN residual concat" indicates concatenation of intermediate GCN layer outputs as a residual representation.

| Model | Backbone | Graph construction/ data | Implementation | Decouple topo & seq | Explicit edge semantics | ResGCN residual concat | Preprocess | Positioning |
|---|---|---|---|---|---|---|---|---|
| Devign | GGNN | Private, tool-dependent | Manual (C/custom) | No | No | No | JIT/ unspecified | Gated GNN for code graphs |
| GraphCodeBERT | Transformer | Large-scale corpus | PyTorch | No (implicit fusion) | No (token/data-flow driven) | N/A | AOT (pretraining) | Large-scale code pretraining |
| EA-RGCN | Dual-channel (GCN+ Transformer) | Private (unverified) | Manual PyTorch | Yes | Yes (edge sequences) | No/not specified | JIT/ unspecified | Dual-channel paradigm |
| PHSCS | RGCN/HAN | Public subset | PyTorch | N/A (hetero focus) | No | N/A | JIT/ unspecified | Heterogeneous graph modeling |
| HAM | BiGRU+ Transformer | Public subset | PyTorch | Yes (sequence-only) | No (token sequence) | N/A | AOT/ unspecified | Sequence-centric hybrid attention |
| VulGNN-SG | Dual-channel (ResGCN+ Transformer) | Public (Etherscan-verified) | PyTorch Geometric (PyG) | Yes | Yes (BFS/DFS edge paths) | Yes (layer-wise concat) | AOT | Reproducible benchmark framework |

graph topology is largely implicit (e.g., via input-level fusion) rather than an explicit structure encoder driven by graph operators.

In this context, the dual-channel paradigm introduced by EA-RGCN provides an important design cue: it explicitly decouples and processes graph topology (via a GCN) and sequential semantics (via a Transformer) in parallel, facilitating complementary information integration.

VulGNN-SG follows this core dual-channel idea; however, we do not claim the dual-channel concept itself as our novelty. Instead, our contributions are primarily methodological and empirical: (i) a transparent, compilation-verified data curation workflow with traceable labels; (ii) an AOT preprocessing and training pipeline that makes large-scale cross-validation feasible and reproducible; and (iii) a controlled traversal study that treats BFS/DFS as an analyzable modeling factor rather than an incidental implementation choice. In addition, we train Word2Vec on a curated global corpus to provide stable node-feature initialization. Accordingly, VulGNN-SG is positioned as a reproducible reference implementation for evaluating traversal-aware dual-channel GNN detectors under consistent data-quality control and implementation conditions.

### 5.2.2. Results and interpretation

The aggregated results of VulGNN-SG+ and all *executable* competing methods across four vulnerability types under different traversal strategies (DFS/BFS) are reported in Tables 6 and 7. The two tables compare VulGNN-SG+ with several SOTA approaches and representative baselines. While F1-score is used as the primary indicator, we also report Accuracy, Precision, Recall, and AUC (whenever continuous prediction scores are available) to reflect overall discriminability and robustness under potential class imbalance.

Overall, VulGNN-SG+ exhibits consistent advantages in most tasks. Specifically, BFS-VulGNN-SG+ achieves F1 scores of 89.22%, 89.84%, 81.81%, and 93.79% on Reentrancy, Timestamp Dependency, Arithmetic, and Unchecked Low Calls, respectively. It outperforms all baseline models and surpasses some prior SOTA methods on selected tasks, indicating stable local-neighborhood aggregation and discrimination. Meanwhile, DFS-VulGNN-SG+ remains highly competitive on vulnerability types that are more dependent on inter-procedural or deep control-flow dependencies. On Unchecked Low Calls, it attains an F1 of 93.54%, which is the best among all DFS-based settings, suggesting effective modeling of deep path-related semantics.

Moreover, VulGNN-SG+ delivers consistently strong performance under both BFS and DFS configurations, supporting the effectiveness of the proposed framework and its competitiveness with SOTA systems. In particular, on Unchecked Low Calls, the BFS configuration

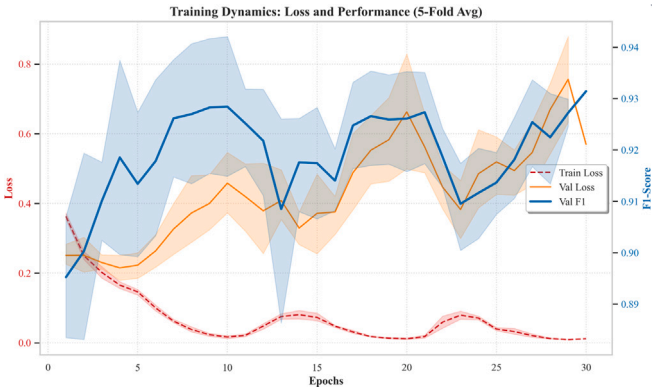



**Fig. 12.** Training dynamics of VulGNN-SG (5-fold average).

achieves 93.79% F1 and 98.61% AUC, yielding the best result within our comparison scope. Importantly, our contribution is not limited to improved scores; we also provide a systematic re-examination and improved implementation of the DFS strategy. Our DFS-VulGNN-SG+ attains an F1 of 89.72% on Reentrancy, substantially outperforming SliSE (78.65%), SC-Defender (86.45%), and VSCL (61.00%), thereby providing direct evidence for the effectiveness of DFS-style modeling.

Fig. 12 reports the training dynamics of VulGNN-SG, averaged over 5-fold cross-validation. As the number of epochs increases, the training loss consistently decreases, while the validation F1-score rises rapidly in the early stage and then stabilizes, indicating effective optimization and convergence with good generalization.

Fig. 13 summarizes the performance of VulGNN-SG+ over the four vulnerability types and contrasts it with representative SOTA methods.

As shown in Fig. 13, the figure comprises: (a) the average F1 bar chart; (b) the multi-task performance trends; (c) the heatmap of $\Delta$F1 relative to our model; and (d) the scatter plot of VulGNN-SG+ under BFS and DFS.

(1) In panel (a), VulGNN-SG+ (BFS+DFS) attains an average F1 of 87.4%, ranking first among all methods. Relative to PHSCS (80.0%) and HAM (83.7%), the gains are +8.5% and +4.3%, respectively, surpassing SOTA models that adopt a single traversal or conventional GNN architectures. The dual-traversal EA-RGCN reaches only 66.8% average F1, 23.6% below VulGNN-SG+, indicating that adding traversal alone does not ensure improvements and may be constrained by weak synergy between traversal policy and structural awareness. Although

**Table 6**
Performance comparison on Reentrancy and Timestamp dependency (%).

| Category | Method | Reentrancy | | | | | Timestamp dependency | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Acc | Pre | Rec | F1 | AUC | Acc | Pre | Rec | F1 | AUC |
| **Ours** | BFS-VulGNN-SG+ | 89.35 | 86.71 | 91.87 | 89.22 | 96.32 | 89.14 | 86.60 | 93.33 | 89.84 | 95.19 |
| | DFS-VulGNN-SG+ | 89.91 | 87.67 | 91.87 | 89.72 | 96.52 | 84.57 | 81.82 | 90.00 | 85.71 | 93.53 |
| DFS-based | DFS-EA-RGCN [21] | 48.95 | 37.85 | 52.19 | 40.09 | 48.74 | 50.69 | 45.10 | 59.61 | 46.70 | 48.65 |
| | SliSE [32] | – | 72.16 | 86.42 | 78.65 | – | – | – | – | – | – |
| | SC-Defender [33] | – | 81.43 | 92.12 | 86.45 | – | – | – | – | – | – |
| | VSCL [34] | 86.00 | 97.00 | 44.00 | 61.00 | – | – | – | – | – | – |
| BFS-based | BFS-EA-RGCN [21] | 94.00 | 94.92 | 93.96 | 94.42 | 98.86 | 91.98 | 95.06 | 91.66 | 93.35 | 96.01 |
| | CGE [35] | 89.15 | 85.24 | 87.62 | 86.41 | 91.00 | 89.02 | 87.41 | 88.10 | 87.75 | 92.00 |
| | DR-GCN [36] | 81.47 | 72.36 | 80.89 | 76.39 | 88.00 | 78.68 | 71.29 | 78.91 | 74.91 | 82.00 |
| Non-traversal | PHSCS [30] | 99.90 | – | – | 98.10 | 100.00 | 95.00 | – | – | 79.30 | 97.70 |
| | HAM [31] | 93.36 | 91.58 | 96.64 | 94.04 | – | 85.62 | 87.03 | 88.69 | 87.85 | – |
| | EGA [37] | 78.00 | 79.00 | 59.00 | 67.00 | – | 94.00 | 84.00 | 44.00 | 58.00 | – |

**Table 7**
Performance comparison on Arithmetic and Unchecked Low Calls (%).

| Category | Method | Arithmetic | | | | | Unchecked Low Calls | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Acc | Pre | Rec | F1 | AUC | Acc | Pre | Rec | F1 | AUC |
| **Ours** | BFS-VulGNN-SG+ | 81.55 | 82.40 | 81.24 | 81.81 | 89.81 | 93.88 | 95.84 | 91.82 | 93.79 | 98.61 |
| | DFS-VulGNN-SG+ | 82.05 | 80.51 | 85.57 | 82.97 | 90.12 | 93.78 | 97.94 | 89.52 | 93.54 | 98.48 |
| DFS-based | DFS-EA-RGCN [21] | 51.78 | 45.69 | 58.53 | 49.47 | 52.34 | 43.65 | 30.57 | 53.75 | 35.67 | 44.13 |
| | SliSE [32] | – | – | – | – | – | – | – | – | – | – |
| | SC-Defender [33] | – | – | – | – | – | – | – | – | – | – |
| | VSCL [34] | – | – | – | – | – | – | – | – | – | – |
| BFS-based | BFS-EA-RGCN [21] | 90.47 | 91.16 | 89.00 | 90.03 | 96.20 | 83.33 | 81.48 | 89.50 | 84.65 | 89.15 |
| | CGE [35] | – | – | – | – | – | – | – | – | – | – |
| | DR-GCN [36] | – | – | – | – | – | – | – | – | – | – |
| Non-traversal | PHSCS [30] | 99.00 | – | – | 66.70 | 99.10 | 98.30 | – | – | 75.90 | 93.90 |
| | HAM [31] | 80.85 | 79.65 | 71.89 | 75.57 | – | 82.56 | 81.24 | 73.46 | 77.15 | – |
| | EGA [37] | 83.00 | 83.00 | 69.00 | 75.00 | – | 93.00 | 87.00 | 41.00 | 56.00 | – |

"–" indicates results that are not available because the corresponding baseline is not executable under our unified pipeline or does not provide continuous prediction scores required for ROC/AUC computation.

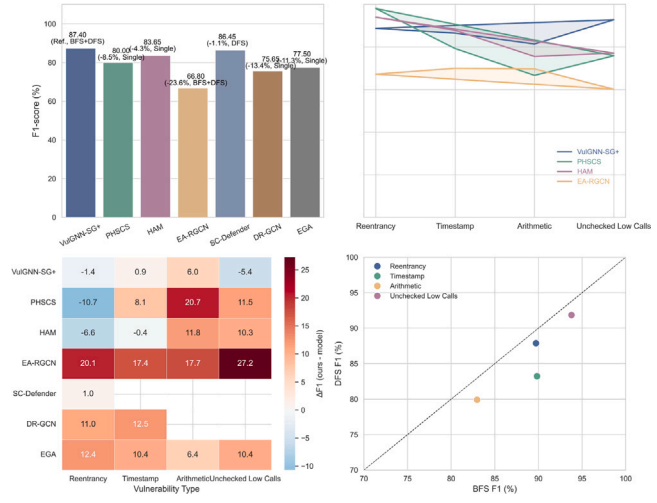



**Fig. 13.** Performance of the VulGNN-SG+ framework and comparison with representative SOTA methods.

SC-Defender achieves 86.5% on a single task under DFS, its overall results remain inferior, suggesting that our systematic DFS design offers better generalization and task consistency.

(2) In panel (b), the F1 trends across four vulnerability types show that VulGNN-SG+ maintains high and stable scores for Reentrancy, Timestamp, Arithmetic, and Unchecked Low Calls, with its curve consistently on top, indicating strong adaptability. PHSCS and HAM exhibit clear declines on Arithmetic and Unchecked Low Calls, revealing modeling bias and limited robustness. EA-RGCN fluctuates more severely, with a collapse on Unchecked Low Calls, confirming that naive traversal fusion is insufficient to capture semantic complexity.

(3) The $\Delta$F1 heatmap in panel (c) shows that, except for VulGNN-SG+, competing models present evident deficits on multiple tasks. Relative to PHSCS and HAM, our model achieves gains up to +20.7% on Arithmetic and +11.5% on Unchecked Low Calls. EA-RGCN is outperformed on all tasks, with $\Delta$F1 up to +27.2%. These results indicate superior semantic modeling and context-dependent vulnerability reasoning of VulGNN-SG+ in complex settings.

(4) Panel (d) shows the correspondence between BFS and DFS F1 for VulGNN-SG+. Points lie near the diagonal, indicating strong positive concordance. For Unchecked Low Calls, BFS reaches 93.8% while DFS remains at 91.9%, demonstrating complementary and stable behavior of breadth- and depth-first traversal within the framework. This corroborates the effectiveness of the proposed dual-traversal structural awareness for multi-task transfer and contextual vulnerability modeling.

Overall, VulGNN-SG+ achieves state-of-the-art results across all four tasks, leading mainstream models by 4%–23% on average. The BFS+DFS strategy enhances robustness and structural diversity. Compared with traditional path analysis, single-traversal GNNs, and program slicing methods, our framework attains new SOTA levels in both accuracy and generalization.

### 5.2.3. *Reassessment and mechanistic analysis of the DFS strategy*

Integrating Tables 6–7 and Fig. 13, we provide a controlled re-examination of the DFS traversal strategy under a unified implementation and evaluation protocol. Beyond reporting strong overall accuracy, the analysis clarifies when depth-oriented traversal yields benefits and

when it does not. Our DFS model attains an F1 of 89.72% on Reentrancy, surpassing representative methods that adopt equivalent traversal ideas, including the slicing-based SliSE (78.65%), the AST-recursive SC-Defender (86.45%), and VSCL with explicit DFS (61.00%).

The superiority stems from a robust implementation of DFS. Compared with the internal baseline DFS-EA-RGCN, the F1 increases from 40.09% to 89.72%, confirming a substantial improvement. These internal and external results indicate that DFS is a traversal with strong expressive potential. Prior underperformance may be influenced by implementation/protocol differences rather than flaws in the strategy itself.

After confirming that both BFS and DFS can reach SOTA performance and that DFS achieves independent superiority, we conduct a fine-grained strategy analysis to address RQ3. The performance gap between the two traversal strategies mainly stems from their different abilities to capture the critical semantic paths required by different vulnerability types. This mechanism can be more intuitively understood by revisiting Fig. 2 in Section 3.2.

(1) Local structure-sensitive vulnerabilities. On Unchecked Low Calls, BFS achieves a higher F1 (93.79%) than DFS (93.54%). This result suggests that detecting such vulnerabilities often relies primarily on local structural cues in the AST neighborhood, e.g., whether requisite safety checks are present around critical *call* nodes. BFS provides Edge_Attention with shorter, semantically related edge sequences, improving capture of local structural cues.

(2) Global semantics-sensitive vulnerabilities. For Arithmetic and Reentrancy, which require long-range dependency understanding, DFS matches BFS and slightly exceeds it on Arithmetic (82.97% vs. 81.81%). These cases depend on cross-statement def-use chains or state-update paths. As visualized in Fig. 2, DFS traversal, by virtue of its depth-oriented exploration, is more likely to progress continuously along a data-flow path (DFG path, red dashed line ❶–❷–❸). The resulting sequence therefore preserves a compact *def→use* dependency chain, which is crucial for identifying potential arithmetic overflow risks. In contrast, BFS traversal tends to follow a control-flow path (CFG path, blue dashed line ①–②–③) or to expand in breadth on the AST. Taking `a=b+c` as an example, BFS typically explores other syntactic neighbors of `b` and `c` before reaching the Assignment node. This behavior inserts many task-irrelevant nodes between the key dependencies, thereby diluting the semantic signal available for representation learning.

In summary, our results indicate that DFS should not be dismissed as inherently ineffective for vulnerability detection. Under controlled data fidelity and a consistent implementation, DFS can be competitive and may be preferable for vulnerability types dominated by long-range def-use or deep control-flow dependencies. These findings provide practical guidance for selecting traversal policies in traversal-aware graph learning pipelines. We further discuss hybrid BFS+DFS fusion (representation-level fusion beyond ensemble averaging) as a promising extension; implementing and ablation-testing such fusion is left for future work.

### 5.3. Ablation and hyperparameter tuning

#### 5.3.1. Comparative methods

**(1) VulGNN-SG (single-model baseline).** Under 5-fold cross-validation, we independently train five models and evaluate each on the held-out final test set. We report the mean and standard deviation of F1 across the five models to assess overall generalization and stability under different data splits.

**(2) VulGNN-SG+ (ensemble).** Using the same 5-fold setting, we average the predicted probabilities of the five independently trained models on the final test set and compute a single deterministic metric from the ensemble output. This design captures the aggregate benefits of ensembling in improving performance, consistency, and robustness.

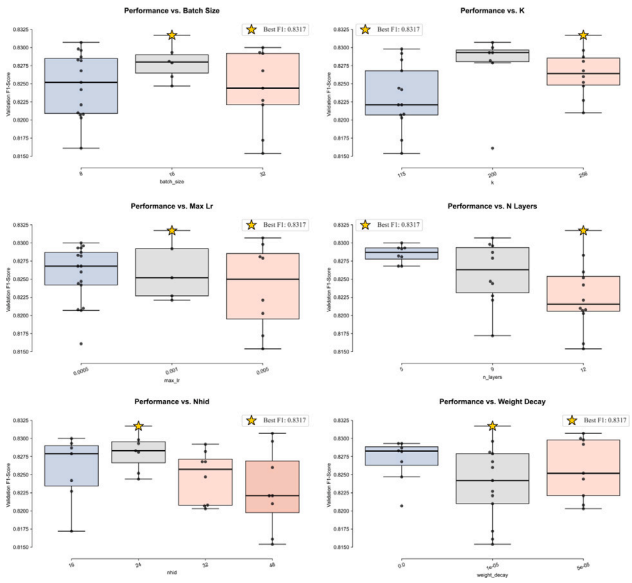



**Fig. 14.** Hyperparameter tuning analysis.

#### 5.3.2. Hyperparameter configuration

To ensure fair evaluation and comparability, we conduct systematic tuning on the VulGNN-SG baseline. Tuning is performed on the largest and most structurally complex dataset, *Arithmetic*, to optimize parameters under the most challenging scenario. We define a comprehensive search space and run 30 random searches. The space includes model width $n_{\text{hid}}$ (16–48), depth $n_{\text{layers}}$ (5–12), pooling range $k$ (115–256), batch size (8–32), L2 regularization $weight\_decay$ (0–$5 \times 10^{-5}$), and maximum learning rate $max\_lr$ (0.0005–0.005).

Each configuration is evaluated by its best F1 on the validation split of the tuning set. Results are summarized in Fig. 14. We select the best configuration (F1 = 83.17%) as fixed hyperparameters for VulGNN-SG across all four vulnerability tasks.

Drawing on empirical insights and tuning results, we adopt the hyperparameter configuration in Table 4 to obtain near-optimal performance across all evaluation tasks.

### 5.4. Ensemble vs. single-channel performance (RQ2)

This section addresses RQ2: whether the proposed VulGNN-SG+ ensemble delivers stable and significant gains over the strong single-model baseline VulGNN-SG.

Table 8 reports results on four vulnerability types under two edge configurations. VulGNN-SG is summarized as mean ± std across five folds, reflecting stability under varying splits. VulGNN-SG+ is reported as a single deterministic metric based on averaged predictive probabilities on the held-out test set.

Across all eight settings, VulGNN-SG+ consistently outperforms VulGNN-SG, with F1 gains between 1.10% and 3.06%. To assess statistical reliability, we conduct one-sample $t$-tests comparing the distribution of single-model scores to the ensemble score for each task. Six of the eight settings show significant improvements ($p < 0.05$).

These findings suggest that aggregating independently trained models from different folds—each with distinct inductive biases—enhances generalization and stability. The results support model ensembling as a robust and broadly applicable performance enhancement.

To further illustrate the advantage of VulGNN-SG+, Fig. 9 and 10 present ROC curves and confusion matrices for two representative tasks. Unchecked Low Calls (BFS) yields the best overall performance, while Reentrancy (DFS) showcases marked and stable gains in a challenging scenario.

**Table 8**
Comparison of VulGNN-SG+ (ensemble) and VulGNN-SG (single-model average) across tasks and edge configurations. VulGNN-SG is reported as mean ± std over five folds; VulGNN-SG+ is the deterministic ensemble output on the held-out test set.

| Vulnerability–Edge | VulGNN-SG (F1%) | VulGNN-SG+ (F1%) | *p*-value | AUC | Accuracy | Precision/Recall |
|---|---|---|---|---|---|---|
| Arithmetic–BFS | 79.38 ± 1.50 | 81.81 | <0.05 | 89.81 | 81.55 | 82.40/81.24 |
| Arithmetic–DFS | 79.91 ± 0.67 | 82.97 | <0.01 | 90.12 | 82.05 | 80.51/85.57 |
| Reentrancy–BFS | 87.77 ± 0.44 | 89.22 | <0.01 | 96.32 | 89.35 | 86.71/91.87 |
| Reentrancy–DFS | 87.87 ± 1.40 | 89.72 | >0.05 | 96.52 | 89.91 | 87.67/91.87 |
| timestamp_dependency–BFS | 88.21 ± 0.89 | 89.84 | <0.05 | 95.19 | 89.14 | 86.60/93.33 |
| timestamp_dependency–DFS | 83.23 ± 3.78 | 85.71 | >0.05 | 93.53 | 84.57 | 81.82/90.00 |
| unchecked_low_calls–BFS | 92.69 ± 0.67 | 93.79 | <0.05 | 98.61 | 93.88 | 95.84/91.82 |
| unchecked_low_calls–DFS | 91.86 ± 0.90 | 93.54 | <0.05 | 98.48 | 93.78 | 97.94/89.52 |

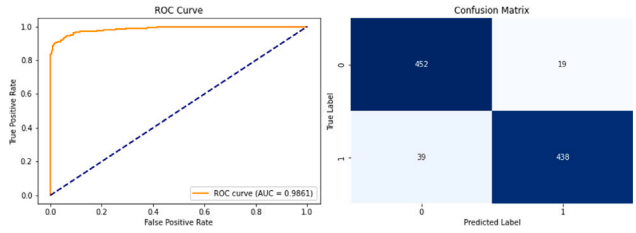



**Fig. 15.** Unchecked Low Calls (BFS): ROC curve and confusion matrix.

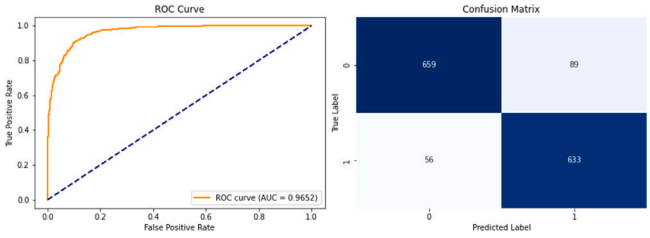



**Fig. 16.** Reentrancy (DFS): ROC curve and confusion matrix.

As shown in Fig. 15, for the *Unchecked Low Calls* task, VulGNN-SG+ attains an AUC of 0.9861. The ROC curve is tightly aligned with the upper-left corner, indicating near-perfect separability between positive and negative classes. The corresponding confusion matrix corroborates this result: among 948 test samples, both false positives and false negatives are rare, demonstrating an excellent balance between precision and recall.

As shown in Fig. 16, on the semantically complex *Reentrancy* task, the ensemble model attains an AUC of 0.9652. The confusion matrix over 1437 test samples indicates a well-balanced trade-off between precision and recall, confirming its effectiveness for detecting vulnerabilities with complex semantics. These visualizations provide direct support for the core metrics reported in the tables.

To assess practical usability, we analyze the computational overhead of the proposed models, as summarized in Table 9. For the Unchecked Low Calls task, a single fold of training for VulGNN-SG takes approximately 0.5 h on average. Performing 5-fold cross-validation for a more robust single-model evaluation requires about 2.5 h of serial computation in total.

A key advantage of the proposed VulGNN-SG+ ensemble is that it incurs *zero additional training cost*. Specifically, the ensemble directly reuses the five models obtained as by-products of the 5-fold cross-validation procedure, without any extra training. The primary overhead is shifted to inference. As shown in Table 9, VulGNN-SG+ increases inference latency by roughly 5× compared with a single model (about 10 s vs. about 2 s), because it performs five forward passes on the test set and averages the predictions.

### 5.5. Ablation study

We evaluate the role and necessity of each component in the dual-channel architecture of VulGNN-SG by comparing the full model with two variants: a *GCN-only* model that retains only the graph-representation branch, and an *Attention-only* model that retains only the edge-semantic branch. Experiments are conducted on two representative tasks with different semantic complexity. Results are shown in Table 10.

Removing either channel causes a marked drop in performance, indicating that the two branches capture non-redundant evidence. The GCN branch primarily contributes topology-driven dependency aggregation, whereas the attention-based edge branch contributes order-sensitive contextual cues along traversal-derived sequences. Their fusion therefore improves prediction by jointly accounting for structural reachability and path-conditioned semantics, rather than assuming either source alone is sufficient.

Channel importance is task-dependent. For *Unchecked Low Calls* (local, structural), the Attention-only model clearly outperforms GCN-only, highlighting the dominance of edge-sequence semantics in recognizing local patterns. For *Arithmetic* (global, semantic), the contributions are more balanced.

These results validate the rationale and necessity of the dual-channel design and reveal an intrinsic linkage between model components and vulnerability semantic complexity.

### 5.6. Deployment-oriented positioning example (literature-grounded)

To further clarify how the proposed pipeline aligns with artifact-centered engineering workflows, we provide an illustrative, literature-grounded positioning example in a blockchain-enabled construction supply-chain context (Fig. 17). The purpose of this subsection is not to claim an implemented end-to-end construction management or payment platform; rather, it demonstrates where an assurance gate based on compilation-verified semantic-graph screening could be inserted to support knowledge-intensive review tasks.

The example is a payment smart contract that may govern the automated release of funds to subcontractors based on as-built progress data captured through reality-capture technologies [38]. Because deployed contracts are immutable and execute autonomously via the Ethereum Virtual Machine (EVM), any latent vulnerability can persist and propagate into operational payment and project-control processes. From an engineering-informatics perspective, pre-deployment inspection is therefore a knowledge-intensive assurance task: reviewers must quickly reason about control- and data-dependencies, cross-function interactions, and deployment-time risk under limited time budgets—analogous to design-review and code-certification workflows in other artifact-centered engineering domains.

In construction supply-chain payment workflows, progress valuations can be transmitted on-chain, while progress evidence may remain off-chain (e.g., in distributed storage) and be referenced through standardized interfaces. In this setting, the smart contract functions as an executable engineering artifact whose failure modes (e.g., timestamp dependence in release conditions) can translate into workflow-level

**Table 9**
Computational cost comparison on the Unchecked Low Calls task.

| Stage | Model | Avg. time | Notes |
|---|---|---|---|
| Training | VulGNN-SG (5-fold CV) | ~2.5 h (total) | Total cost to train five independent models. |
| | VulGNN-SG+ (ensemble) | 0 | Reuses CV-produced models; no extra training. |
| Inference | VulGNN-SG (single) | ~2 s | One pass of prediction over the entire test set. |
| | VulGNN-SG+ (ensemble) | ~10 s | Five passes and averaging over the test set. |

**Table 10**
Ablation of the VulGNN-SG dual-channel architecture on two representative tasks.

| Model variant | Unchecked Low Calls (Local, Structural) | | | | | Arithmetic (Global, Semantic) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | F1 | AUC | Accuracy | Precision | Recall | F1 | AUC | Accuracy | Precision | Recall |
| GCN-only | 76.19 | 81.38 | 76.95 | 77.81 | 79.03 | 72.55 | 82.51 | 72.89 | 77.04 | 68.74 |
| Attention-only | 87.13 | 93.98 | 88.38 | 87.58 | 89.28 | 75.09 | 84.97 | 74.88 | 77.44 | 73.15 |
| VulGNN-SG (Dual-Channel) | 92.69 | 98.06 | 92.70 | 93.65 | 91.82 | 79.38 | 88.05 | 79.15 | 80.17 | 78.83 |

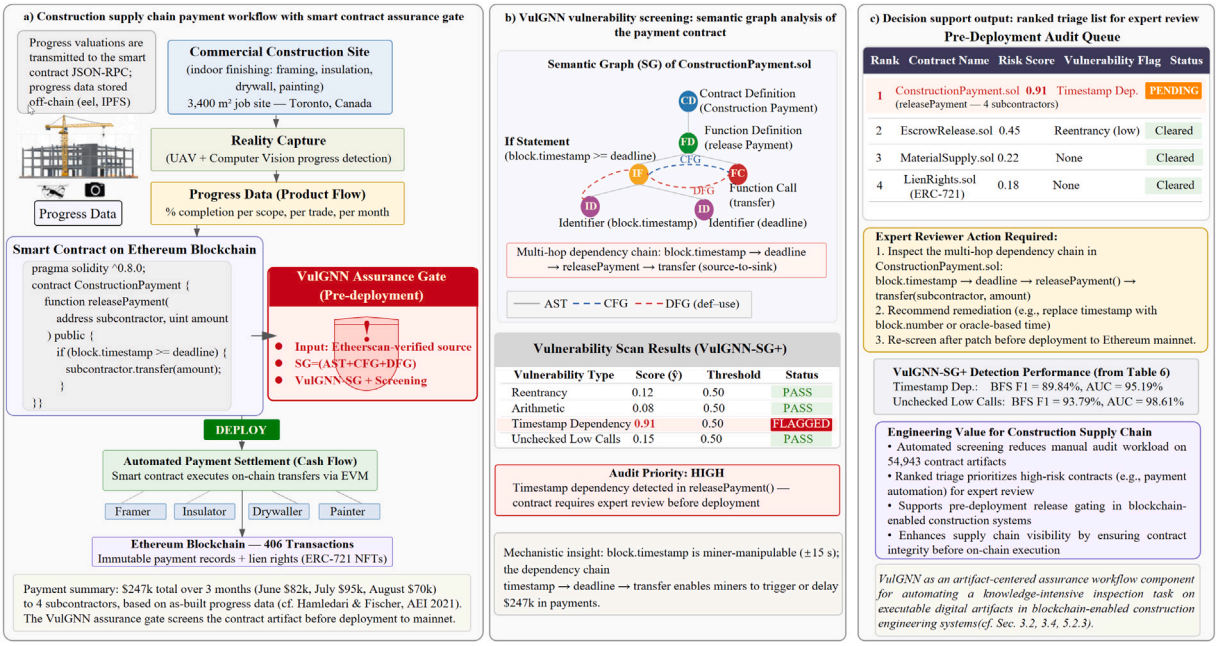



**Fig. 17.** Literature-grounded positioning example of an artifact-centered assurance gate in a blockchain-enabled construction workflow (illustrative). The figure illustrates how a compilation-verified semantic-graph screening step (VulGNN) could be inserted before mainnet deployment to support audit triage and release gating. This paper contributes the screening pipeline and controlled evidence on Etherscan-verified contract artifacts; it does not implement a full construction/payment platform.

risks. Fig. 17 situates VulGNN as a pre-deployment assurance gate for this artifact-centered workflow interpretation of our empirically validated pipeline (Sections 3–5): (i) ingest an Etherscan-verified contract artifact, (ii) enforce compilation and structural validation as in Sections 3.2–3.3, (iii) construct the SG representation (AST+CFG+DFG) as in Sections 3.1–3.3, and (iv) output a risk score and ranked triage signal to support expert review and release gating under the fixed protocol in Section 4.1.2. Importantly, the figure is intended to illustrate this workflow interpretation only, without introducing new functional modules or additional experimental claims beyond the smart-contract datasets evaluated in this paper.

## 6. Conclusion

We address data quality and implementation fidelity in smart contract vulnerability detection and extend a graph neural network (GNN)-based approach. The main contributions are as follows.

(1) We propose *VulGNN*, a high-fidelity experimental framework. Using verified contracts from Etherscan, we build a curated data pipeline that yields high-quality, class-balanced, and reproducible datasets for rigorous evaluation.

(2) We implement *VulGNN-SG*, a high-performance dual-channel baseline for GNN-based vulnerability detection. Within this framework, we engineer a strong single-channel baseline and demonstrate leading performance. The results also uncover the potential advantage of DFS-based graph representations for this task.

(3) We reveal an intrinsic linkage between traversal strategy and vulnerability semantics. Empirical evidence shows that the optimal traversal depends on vulnerability type, suggesting that traversal should adapt to semantic complexity.

(4) We propose the *VulGNN-SG+* ensemble. Built on the validated baseline, a K-fold ensemble delivers stable and significant gains across all four tasks, with F1 improvements up to 3.06%.

This study has several limitations that should be considered when interpreting the results. First, the ground-truth labels are inherited from *smartbugs-results* (generated by existing analyzers), which inevitably introduces label noise; in particular, a negative sample indicates "not flagged" rather than a formally verified absence of the target vulnerability. Second, we focus on four representative vulnerability types and formulate one binary classification task per type; the current setting does not address multi-label scenarios where multiple vulnerabilities

co-exist within the same contract. Third, although our ahead-of-time (AOT) pipeline improves reproducibility under controlled compilation and parsing, the semantic graph construction still depends on the underlying toolchain and graph-building rules; changes in compiler versions, language features, or analysis configurations may affect graph stability and cross-environment generalization. Finally, traversal-derived edge sequences (BFS/DFS) provide a practical and analyzable approximation of path-conditioned semantics, but they do not fully capture all runtime behaviors (e.g., complex inter-contract interactions and dynamic dispatch), which remains an open direction for future work.

Overall, we present a reproducible graph-learning framework for smart-contract vulnerability detection and empirically demonstrate the central roles of data fidelity and traversal-induced semantics. We hope the released pipeline and controlled findings can serve as a practical reference for future traversal-aware vulnerability detectors. While the presented workflow principles (compilation-verified curation, dependency-preserving graph representations, and traceable screening outputs) are broadly applicable to artifact-centered assurance settings, our quantitative evidence is currently established on Etherscan-verified smart-contract artifacts and should be generalized to other engineering artifacts only with dedicated empirical validation.

## CRediT authorship contribution statement

**Xinyi Chen:** Methodology. **Weihua Bai:** Writing – original draft. **Jialing Zhao:** Software. **Tao Huang:** Software. **Huibing Zhang:** Validation. **Teng Zhou:** Project administration. **Keqin Li:** Supervision.

## Declaration of competing interest

The authors declare that they have no conflict of interest.

## Acknowledgments

This work was supported by the Guangxi Natural Science Foundation (No. 2026GXNSFAA00641306), the National Natural Science Foundation (No. 62462021, 62267003, 62177012), Guangxi Key Laboratory of Trusted Software (No. KX202319), the Special Fund for Guangdong Province University Key Field (No. 2023ZDZX3041), the Philosophy and Social Sciences Planning Project of Zhejiang Province (No. 25JCXK006YB), the Hainan Provincial Natural Science Foundation of China (No. 625RC716), Guangdong Basic and Applied Basic Research Foundation (No. 2025A1515010197), and Zhaoqing University Innovation Research Team Grant Project, and University-Level High-Caliber Training Project of Zhaoqing University (No. gcc202607).

## Data availability

Data will be made available on request.

## References

[1] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), in: International Conference on Principles of Security and Trust, Springer, 2017, pp. 164–186.
[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.
[3] C.S. Yashavant, S. Kumar, A. Karkare, Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities, 2022, arXiv preprint arXiv:2202.11409.
[4] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Adv. Neural Inf. Process. Syst. 32 (2019).
[5] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 297–308.
[6] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, IEEE, 2019, pp. 783–794.
[7] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.
[8] Y. Fan, C. Wan, C. Fu, L. Han, H. Xu, VDoTR: Vulnerability detection based on tensor representation of comprehensive code graphs, Comput. Secur. 130 (2023) 103247.
[9] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, 2017, arXiv preprint arXiv:1711.00740.
[10] P.W. Battaglia, J.B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al., Relational inductive biases, deep learning, and graph networks, 2018, arXiv preprint arXiv:1806.01261.
[11] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, Adv. Neural Inf. Process. Syst. 30 (2017).
[12] T. Kipf, Semi-supervised classification with graph convolutional networks, 2016, arXiv preprint arXiv:1609.02907.
[13] S. Brody, U. Alon, E. Yahav, How attentive are graph attention networks?, 2021, arXiv preprint arXiv:2105.14491.
[14] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, 2017, arXiv preprint arXiv:1710.10903.
[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P.S. Yu, A comprehensive survey on graph neural networks, IEEE Trans. Neural Netw. Learn. Syst. 32 (1) (2020) 4–24.
[16] M. Schlichtkrull, T.N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, in: European Semantic Web Conference, Springer, 2018, pp. 593–607.
[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254–269.
[18] J. Feist, G. Grieco, A. Groce, Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB, IEEE, 2019, pp. 8–15.
[19] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, Smartcheck: Static analysis of ethereum smart contracts, in: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 2018, pp. 9–16.
[20] C. Ying, T. Cai, S. Luo, S. Zheng, G. Ke, D. He, Y. Shen, T.-Y. Liu, Do transformers really perform badly for graph representation? Adv. Neural Inf. Process. Syst. 34 (2021) 28877–28888.
[21] D. Chen, L. Feng, Y. Fan, S. Shang, Z. Wei, Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention, J. Syst. Softw. 202 (2023) 111705.
[22] J.F. Ferreira, P. Cruz, T. Durieux, R. Abreu, Smartbugs: A framework to analyze solidity smart contracts, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1349–1352.
[23] U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2vec: Learning distributed representations of code, Proc. ACM Program. Lang. 3 (POPL) (2019) 1–29.
[24] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013, arXiv preprint arXiv:1301.3781.
[25] M. Zhang, Z. Cui, M. Neumann, Y. Chen, An end-to-end deep learning architecture for graph classification, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 32, 2018.
[26] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, 2015, arXiv preprint arXiv:1511.05493.
[27] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J.M. Zhang, Q. Liao, Vulnerability detection with graph simplification and enhanced graph representation learning, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 2275–2286.
[28] B. Lakshminarayanan, A. Pritzel, C. Blundell, Simple and scalable predictive uncertainty estimation using deep ensembles, Adv. Neural Inf. Process. Syst. 30 (2017).
[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).
[30] S. Jiang, Y. Chen, T. Ouyang, X. Zhang, S. Su, Pruning attention heads based on semantic and code structure for smart contract vulnerability detection, IEEE Trans. Dependable Secur. Comput. (2025).
[31] H. Wu, H. Dong, Y. He, Q. Duan, Smart contract vulnerability detection based on hybrid attention mechanism model, Appl. Sci. 13 (2) (2023) 770.
[32] Z. Wang, J. Chen, Y. Wang, Y. Zhang, W. Zhang, Z. Zheng, Efficiently detecting reentrancy vulnerabilities in complex smart contracts, Proc. ACM Softw. Eng. 1 (FSE) (2024) 161–181.
[33] A. Mittal, G. Widjaja, R.D.C. Pecho, R. Kiruba, J.M.F. Roque, A. Chandra, Blockchain based abstract syntax tree to detect vulnerability in IOT-enabled smart contract, in: 2023 Second International Conference on Smart Technologies for Smart Nation, SmartTechCon, IEEE, 2023, pp. 270–275.

[34] F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, L. Khan, VSCL: automating vulnerability detection in smart contracts with deep learning, in: 2021 IEEE International Conference on Blockchain and Cryptocurrency, ICBC, IEEE, 2021, pp. 1–9.

[35] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, X. Wang, Combining graph neural networks with expert knowledge for smart contract vulnerability detection, IEEE Trans. Knowl. Data Eng. 35 (2) (2021) 1296–1310.

[36] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, Q. He, Smart contract vulnerability detection using graph neural networks, in: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, 2021, pp. 3283–3290.

[37] S. HajiHosseinKhani, A novel vulnerable smart contracts profiling method based on advanced genetic algorithm using penalty fitness function, 2024.

[38] H. Hamledari, M. Fischer, Measuring the impact of blockchain and smart contracts on construction supply chain visibility, Adv. Eng. Inform. 50 (2021) 101444.