International Journal of Safety and Security Engineering
Vol. 16, No. 7, July, 2026, pp. 1439-1450
Journal homepage: http://iieta.org/journals/ijsse

Slice-Anchored Multimodal Fusion for Function-Level Smart Contract Vulnerability Detection

Shankar Gugulothu*, Nandhini Malaiyappan

Department of Computer Science, Pondicherry University, Puducherry 605014, India

Corresponding Author Email: shankarsaida@gmail.com

https://doi.org/10.18280/ijsse.160703

Received: 25 April 2026
Revised: 6 July 2026
Accepted: 13 July 2026
Available online: 31 July 2026

Keywords:
smart contract vulnerability, Control Flow Graph, Abstract Syntax Tree, Call Graph, CodeBERT, Graph Convolutional Network

ABSTRACT

Smart contracts are a cornerstone of blockchain applications; however, vulnerabilities within these contracts pose substantial financial and security risks, making early detection before deployment a critical necessity. Traditional rule-based approaches fall short in capturing the complex semantic and structural patterns inherent in smart contract code, limiting their effectiveness as a reliable detection mechanism. To address this problem, we propose a multimodal deep learning approach that is specifically aimed at identifying vulnerabilities in smart contracts. The suggested architecture uses CodeBERT to learn semantic representations in source code and, at the same time, learn structural information in program graphs, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Call Graph (CG). These semantic and structural features are then incorporated together by a slice-anchored fusion mechanism. The proposed approach resulted in an accuracy of 91.35%, a precision of 90.87%, a recall of 92.12% and an F1-score of 91.49%.

1. INTRODUCTION

Smart contracts are self-executing programs that operate on blockchain platforms and automatically carry out predefined actions when specified conditions are satisfied. They enable transactions and agreements between parties without the need for a trusted intermediary by embedding the contractual rules directly into code. Through this mechanism, smart contracts can manage digital assets and enforce business logic within decentralized systems. Despite these advantages, the reliability of smart contracts is strongly dependent on the correctness of their implementation [1]. Since the deployed code becomes immutable on the blockchain, even minor programming errors can lead to serious security problems. Such weaknesses in the contract logic are commonly referred to as vulnerabilities and can potentially expose blockchain applications to financial and operational risks [2]. A smart contract is immutable, unlike traditional software. Any defect in its code will therefore be constantly on display and, in many instances, very costly to its users. This fact has put the field of vulnerability detection at the centre of blockchain security research, since the consequences of neglecting even a small coding bug can be astronomical [3]. This requirement is fulfilled by the current research, which is based on the systematic evaluation of existing detection instruments, including both conceptual and practical testing of their effectiveness in securing blockchain applications. Despite the large number of smart contract vulnerability detection tools, it is still hard to research this sphere because the approaches are extremely diverse, and their assessment can be performed on different grounds [4]. To deal with this problem, a thorough review of vulnerability detection is performed in this paper.

In the previous decade, many studies for detecting vulnerability in smart contracts used techniques such as symbolic execution, data-flow analysis, and static analysis, laying important groundwork, yet each carries inherent limitations. Symbolic execution, while thorough in theory, suffers from the well-documented path explosion problem, where computational demands grow exponentially with contract complexity [5]. Data-flow analysis, though useful for tracing information movement, frequently fails to achieve complete behavioral coverage, leaving certain execution scenarios unexamined [6, 7]. Statistical approaches, including frequency-based anomaly detection and probabilistic risk modelling, have also been explored; however, their reliance on predefined thresholds and historical distributions makes them poorly suited for detecting attack patterns [8, 9]. ML-based models are also extensively used for efficient understanding of vulnerability patterns [10]. All of these conventional techniques rely heavily on hand-written rules and domain knowledge. In order to deal with these challenges, many studies have used DL based models. DL allows the model to learn useful patterns in previously examined contracts rather than encoding vulnerabilities in hand-written logic. This transformation makes the detection process much more dynamic and scalable. It enables the system to identify complex, deeply entrenched vulnerability patterns that are difficult to capture with traditional and statistical methods [11-13].

Although deep learning-based methods reduce reliance on manually-defined rules, most of them process smart contracts as flat token sequences, which limits the extraction of structural information including control flow, syntax, and dependency relationships. While such approaches can capture

parts of the contextual signal, they often fail to reflect the detailed structural connections present in smart contracts syntactic structure, control flow, data dependencies, and inter-function interactions and vulnerability-relevant cues are diluted by surrounding boilerplate, weakening detection effectiveness.

To address these limitations, this paper proposes a fine-grained, statement-level multimodal approach to smart contract vulnerability detection. Rather than treating each function as a single unit, the function is first decomposed into a program slice driven automatically by vulnerability sinks-external calls, low-level calls, ether transfers, Tx.origin (TX) uses, self destructs, delegate calls, and authorization checks. The slice retains only the statements whose control- or data-dependence reaches a sink, eliminating redundant code. Unlike prior slicing-based detectors such as DeepFusion, which rely on a separate hand-crafted slicer per vulnerability class (PSER for Reentrancy (RE), PSETX for TX, PSEI for integer bugs, and so on), and graph-based detectors such as CGE/AMEVul Detector, which depend on per-vulnerability expert "core node" rules to focus their graph, our procedure is a single vulnerability-agnostic backward-slicing step that generalizes across the DASP taxonomy.

From each slice, CodeBERT extracts a semantic representation of the surviving source tokens, providing contextual and lexical descriptions of vulnerability-relevant code. In parallel, the same slice is projected into three structural views an Abstract Syntax Tree (AST), a Control Flow Graph (CFG), and a Call Graph (CG) pruned to the sliced statements with transitive edge reconnection so that flow relationships are preserved across removed nodes. Each view is encoded by a Graph Convolutional Network (GCN) to capture syntactic structure, execution behavior, and inter-function dependencies.

Crucially, the semantic and structural views are not combined as globally-pooled vectors, as in prior multimodal detectors such as DeepFusion, VulnSense, and ContractShield. Instead, a slice-anchored fusion mechanism aligns the CodeBERT token span of each sliced statement with its corresponding CFG node, producing a fused per-statement vector that binds what the statement says to where it sits in the control- and data-flow graph. The per-statement representations are then aggregated and combined with the AST and CG views to produce the final classification. This statement-level alignment preserves the code-to-graph correspondence that global pooling discards, and is the central methodological novelty of this work. The primary contributions of this study are:

•We apply program slicing to automatically isolate vulnerability-relevant code statements from smart contracts before any feature extraction, reducing noise and directing the model's attention toward the execution paths most likely to contain security flaws.

•We construct three graph representations-CFG, AST, and CG-directly from the sliced code and apply GCNs to extract rich structural features that capture control flow, syntactic structure, and inter-function dependencies.

•We use CodeBERT to extract semantic representations from the sliced source code, capturing the contextual meaning and relationships between code tokens in vulnerability-relevant statements.

•We propose a slice-anchored fusion mechanism that aligns each sliced statement's semantic representation from CodeBERT with its corresponding node in the CFG, fusing semantic and structural information at the statement level rather than combining them as a single global vector. This preserves the connection between what each statement means and where it sits in the program flow, producing a more precise contract representation.

•We evaluate our approach on a real-world dataset of 6,698 Ethereum smart contracts spanning four vulnerability classes-RE, Access Control (AC), TX, and non-vulnerable. Experiments demonstrate that slice-anchored fusion achieves an accuracy of 91.35%, outperforming weighted fusion and yielding consistent gains across all vulnerability classes.

The remainder of this paper is organized as follows. Section 2 reviews related work in smart contract vulnerability detection. Section 3 outlines the types of smart contract vulnerabilities. Section 4 presents the proposed approach and its underlying methodology. Section 5 describes the experimental setup. Section 6 analyzes and discusses the results in comparison with existing methods. Section 7 concludes the paper.

2. RELATED WORK

Smart contract vulnerability detection has evolved through three broad phases: rule-based static analysis, sequential machine learning, and graph-based deep learning. Each phase addressed a limitation of the previous one, and each introduced new limitations of its own. We organize the literature around this progression and identify the gap our work targets.

Early detection tools used symbolic execution, fuzz testing, and static analysis to apply hand-crafted vulnerability patterns to contract source or bytecode. Symbolic execution approaches such as Oyente and its successors enumerate execution paths against constraint solvers [5], while static-analysis frameworks such as Slither operate on intermediate representations of Solidity source [9], and bytecode-focused analysers such as Mythril combine symbolic execution with taint analysis [8]. Improved bytecode-level CFG construction has continued to refine this line of work [14]. These tools are precise on the patterns they encode but require expert knowledge for every new vulnerability class, suffer from path-explosion on large contracts, and cannot adapt to emerging vulnerability patterns. The first wave of machine-learning approaches addressed this brittleness through learned features over surface representations: S-gram extracted n-gram language models from AST token sequences [15]; ContractWard combined opcode-level feature extraction with classical classifiers and reported strong performance on Ethereum contracts [16] and SoliAudit combined machine learning with grey-box fuzz testing to remove dependence on expert rules [17]. A broader survey of this line is provided in reference [1]. However, all of these methods still treated contracts as flat token or opcode sequences, discarding structural and control flow information that is central to how vulnerabilities actually manifest.

Deep learning enabled automatic feature extraction from raw contract code and removed the manual-feature bottleneck. An optimised CodeBERT variant that extracts vulnerable function segments and leverages CodeBERT's pre-training for classification, achieving high F1 on the SolidiFI benchmark, was proposed [11]. Deng et al. [18] combined CNN and BiGRU networks to extract local and sequential features from contract opcodes and fused these complementary views through a multimodal decision mechanism, and subsequent

deep learning frameworks have continued to refine this line of work with novel architectures and multi-objective training objectives [2, 12, 13]. Yet sequential models share a fundamental limitation: they consume contract source or opcodes as a one-dimensional token sequence and cannot directly represent the control flow, data flow, and inter-procedural relationships that determine whether a vulnerability is exploitable. A function with a *call.value* invocation is vulnerable only when its state update follows the call and no access-control modifier intervenes, a property that depends on graph structure, not on token order.

This limitation motivated graph-based and multimodal approaches that consume CFGs, AST, CGs, and program-dependence graphs (PDGs) through graph neural networks. The progression visible in the table is consistent: from single-graph methods (DR-GCN and TMP on contract graphs [19], CGE extending these with expert patterns [20], MANDO using heterogeneous CFG + CG [21]) to multi-graph fusion (Cai's sliced AST + CFG + PDG joint graph [22], Ma's hierarchical AST + CFG attention [23], Shang's connectivity-enhanced CFG with transformers [24]). Peculiar combined critical data-flow graphs with pre-trained code representations [6], and VulnSense combined a language model with graph neural networks over multiple contract representations [25]. Cai et al. [22] further extended this direction with heterogeneous code-feature learning and automated dataset construction [4]. Across these methods, however, multimodal fusion is performed by independently pooling each modality into a single global vector and then concatenating or attentively weighting those vectors before classification. This pool-then-fuse pattern discards the correspondence between specific source-code statements and the graph nodes that represent them: after pooling, the model can no longer tell which statement contributed which signal. For vulnerability classes whose discriminative pattern is local to a single statement (a missing authorization check, a state update after an external call, a use of TX), this loss of statement-level alignment is precisely where existing multimodal methods leave information on the table.

The gap our work targets is therefore not the absence of multimodality, nor the absence of slicing - both have been explored. It is the absence of statement-level cross-modal alignment in existing multimodal fusion. Our slice-anchored fusion mechanism explicitly aligns CodeBERT [26] token spans to CFG nodes via the slicer's character-span map, performs cross-modal fusion at the statement level, and only then aggregates - inverting the pool-then-fuse ordering used by every method. The remainder of this paper describes this mechanism, its empirical behavior in a controlled within-family ablation against global concatenation and weighted-fusion baselines, and a comparison against reimplemented prior slicing-based detectors on identical data and splits.

3. SMART CONTRACT VULNERABILITIES

This section examines the most critical smart contract vulnerability types, including RE, AC and TX. Each vulnerability is discussed in terms of its root cause and potential impact. Figure 1 illustrates an example of a well-structured smart contract code free from any detectable vulnerabilities.

```
contract Reentrancy {
    mapping(address => uint) balance;

    function withdraw(uint amt) public {
        if(balance[msg.sender] >= amt){
            msg.sender.call{value: amt}("");
            balance[msg.sender] -= amt;
        }
    }
}
```

Figure 1. Smart contract with reentrancy vulnerability

3.1 Reentrancy vulnerability

RE is one of the most well-known vulnerabilities in smart contracts, and it occurs when a contract calls itself before a previous execution completes. This is usually the case when a contract makes an external call to a different address before updating its internal state. If the external contract contains malicious code, it can re-enter the original functionality and execute itself multiple times before the balance or state variable is changed, the Figure 1 depicts the smart contract with RE vulnerability. Therefore, attackers can repeatedly withdraw funds or control contract behavior, leading to significant financial losses. A notable example of a RE-vulnerable attack is the 2016 DAO attack, which resulted in a massive loss of Ether. RE vulnerabilities are also regularly addressed by adopting the checks-and-effects pattern, in which state variables are updated before external calls are made.

3.2 Access control

A vulnerability in AC occurs when important functions in a smart contract are not restricted to authorized parties. In most contracts, sensitive operations, such as transfer of ownership, change of parameters, or transfer of funds, are restricted to the owner of the contract or an administrator. AC vulnerabilities happen due to the lack of authorization checks, incorrect implementation of access modifiers, or improper authentication mechanisms. As a result, an attacker may gain unauthorized privileges and modify critical contract parameters. These risks can be reduced by adopting strong authorization controls and role-based AC policies. Smart contract with AC Vulnerability is shown in Figure 2.

```
contract AccessControl {
    address public owner;

    function changeOwner(address newOwner) public {
        owner = newOwner;
    }
}
```

Figure 2. Smart contract with access control vulnerability

3.3 Tx.origin

The TX vulnerability occurs when a smart contract uses TX for authentication instead of *msg.sender*. While TX always refers to the original external account that initiated the transaction chain, *msg.sender* refers only to the immediate caller. When a contract relies on TX for AC, as shown in Figure 3, any intermediate contract invoked by the legitimate owner can exploit this to impersonate that owner and bypass authorization checks. A typical attack involves a malicious contract tricking the owner into initiating a transaction, after which the malicious contract forwards a call to the victim

contract-since TX still holds the owner's address, the authorization passes and the attacker gains unauthorized access. The consequences include unauthorized fund withdrawal, unintended ownership transfers, and complete loss of contract control. To mitigate this vulnerability, developers should always use msg.sender for authentication rather than TX, reserving TX strictly for non-authorization purposes where identifying the original initiator is explicitly necessary.

```
contract TxOriginVulnerable {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function transfer(address payable dest, uint amount) public {
        require(tx.origin == owner, "Not authorized");
        dest.transfer(amount);
    }
}
```

Figure 3. Smart contract with Tx.origin vulnerability

4. PROPOSED WORK

This work presents a function-level multimodal detection framework for identifying smart contract vulnerabilities across four classes: non-vulnerable, RE, AC, and TX. The framework begins by decomposing each function into a vulnerability-focused program slice through a sink-driven backward-slicing procedure.

Relevant sinks including external calls, ether transfers, TX references, self-destruct operations, delegatecall invocations, and authorization checks anchor the slice, and all statements connected to these sinks through control or data dependence are retained while the remaining function body is discarded. This targeted reduction confines feature extraction to code that is structurally relevant to the vulnerability, rather than exposing the model to the full body of noisy contract code.

As illustrated in Figure 4, from each resulting slice, the framework derives parallel representations at two levels. On the structural side, three graph projections a CFG, an AST, and a CG are constructed from the surviving statements with transitive edge reconnection to preserve flow continuity, then processed jointly by GCNs to yield complementary structural embeddings. On the semantic side, CodeBERT encodes the sliced source tokens to produce a context-aware lexical representation. Rather than combining these two streams through a single global pooling step, the framework employs a slice-anchored fusion mechanism: each sliced statement's token span is aligned to its corresponding CFG node, and semantic and structural information are fused at the statement level before aggregation. This preserves the correspondence between code meaning and graph position that global fusion discards. The resulting unified representation is passed to a fully connected classifier to produce the final vulnerability label.

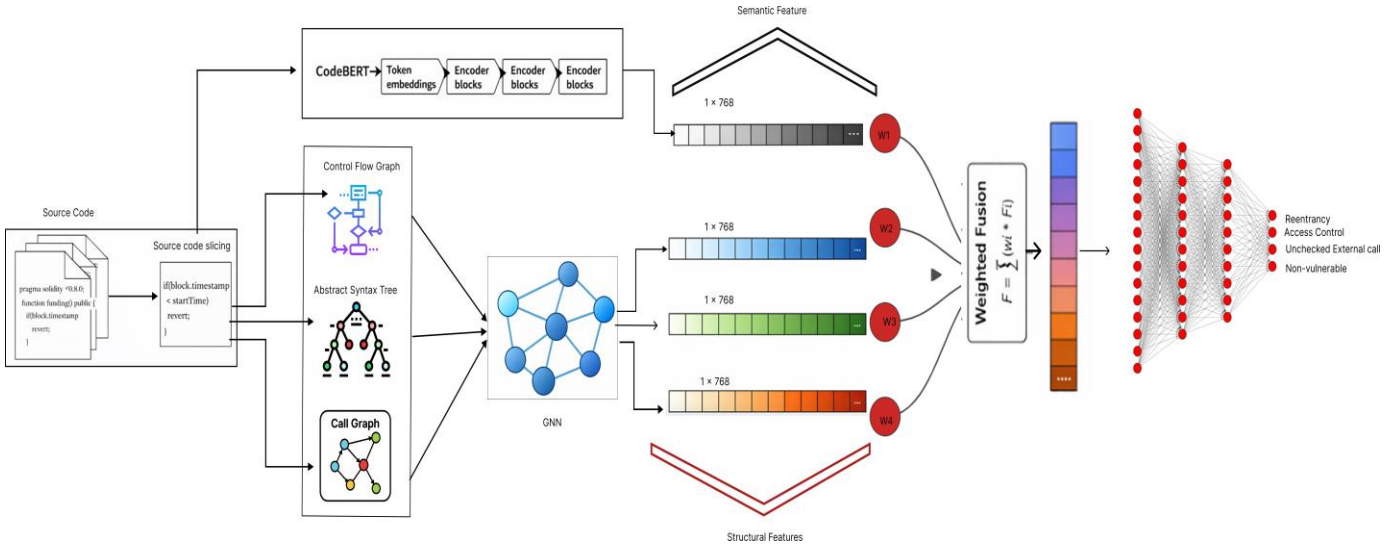

Figure 4. Slice-anchored fusion for smart contract vulnerability detection

4.1 Structural Vulnerability Program Slicing

Program slicing is the process of extracting from a larger program the subset of statements that influence a chosen point of interest, known as the slicing criterion. In the context of smart-contract security, the criterion is naturally taken to be a statement that has the potential to trigger a vulnerability - a call to an untrusted external address, a use of TX for authorisation, a self-destruct, and so on. A smart-contract function typically contains many statements that have no bearing on such operations: arithmetic helpers, accessor methods, event emissions, and other supporting logic that contribute volume but not security signal. Including these statements indiscriminately during feature extraction injects noise into the learned representation and dilutes the features that actually distinguish vulnerable from safe code. The purpose of program slicing in our pipeline is therefore to retain only those statements that are causally connected to a vulnerability-triggering operation, both through the data that flows into it and through the control conditions that gate its execution. In this study, we use the Structural Vulnerability Program Slicing (SVPS) framework to extract vulnerability-related code fragments from the smart contract, as shown in Algorithm 1.

We refer to our procedure as SVPS and apply it independently to every function in the contract. SVPS operates in three stages. The contract is first compiled and statically analysed using Slither with a pragma-matched solc version,

producing the function-level CFG, the AST, and the CG. A set of vulnerability sinks is then identified across the entire contract. These sinks fall into three groups aligned with the vulnerability classes considered in this work: RE-related operations (call, low-level calls, and ether transfers), TX references used in an authorization context, and access-control-sensitive operations (selfdestruct, delegate call, and explicit owner or role checks). The choice of sinks is informed by established taxonomies of smart-contract vulnerabilities and reflects the operations through which the corresponding flaws manifest. Finally, for each function containing at least one such sink, the algorithm performs a backward slice from every sink in that function, traversing both data-dependence relationships-variables whose definitions reach the sink along def-use chains and control-dependence relationships-predicates whose outcome determines whether the sink is reached. The union of statements gathered through these dependence chains, together with the sink statements themselves, constitutes the slice for that function.

Algorithm 1. SVPS

Require: Smart-contract source code C
Ensure: Per-function fragment FragSet = {$Frag_1$, …, $Frag_n$}
1. FragSet ← ∅
2. G^{cfg}, S^{code}, G^{ast} ← CompileAndAnalyse(C), Slither + solc
3. Vset ← IdentifyVulnerabilitySinks(C), sinks across C
4. for all function f ∈ C do
5. F_{sinks} ← Vset ∩ Statements(f)
6. if F_{sinks} = ∅ then
7. Label(f) ← non_vulnerable
8. continue
9. end if
10. SliceStmts ← ∅
11. for all sink s ∈ F_{sinks} do
12. D_{ata} ← BackwardDataDependence(s, f)
13. Dc_{trl} ← BackwardControlDependence(s, f)
14. SliceStmts ← SliceStmts ∪ D_{ata} ∪ Dc_{trl} ∪ {s}
15. end if
16. G^{cfg}_f ← PruneAndReconnect(G^{cfg}, f, SliceStmts), transitive edges
17. G^{ast}_f ← PruneSubtree(G^{ast}, f)
18. G^{c}_f ← RestrictToCallees(G^{c}, f)
19. A_f ← BuildAlignment(G^{cfg}, SourceTokens(f)), node ↔ span ↔ lines
20. Frag_f ← (G^{cfg}_f, G^{ast}_f, G^{c}_f, A_f, SourceText(SliceStmts))
21. FragSet ← FragSet ∪ {Frag_f}
22. end for
23. return FragSet

Once the slice has been computed, three structural views of the function are produced directly in graph space rather than by recompiling text fragments. The CFG is pruned to the slice nodes, and the edges of removed nodes are bypassed through transitive reconnection so that reachability between the surviving statements is preserved. The AST is reduced to the syntactic subtree spanning the slice statements, retaining the hierarchical relationships needed for syntactic feature learning. The CG is restricted to the function under analysis together with its directly invoked callees, providing the inter-function context that influences vulnerability behavior. In addition to these three graph projections, an alignment map is constructed linking each surviving CFG node to its corresponding token span in the source code; this alignment is the structural anchor consumed by the slice-anchored fusion mechanism introduced later in the paper. Functions in which no sink is present are labelled non-vulnerable and bypass the slicing step.

4.2 Semantic representation using CodeBERT

Vulnerability-relevant code fragments are extracted by using the SVPS algorithm. Then, relevant code fragments are passed to CodeBERT, a pre-trained transformer-based model specifically adapted for programming languages [26]. CodeBERT has found wide application in capturing contextual semantics of source code, due to its bidirectional attention models trained on large-scale code corpora.

The code fragment is first broken down into a series of tokens which is denoted by the Eq. (1), where t_i denotes the ith token in the fragment and n represents the total number of tokens in the sequence. Each token is then encoded into a continuous embedding using the CodeBERT embedding layer as shown in Eqs. (2) and (3).

$$T = \{t_1, t_2, \dots, t_n\} \tag{1}$$

$$e_i = E(t_i), e_i \in R^d \tag{2}$$

$$E_T = \{e_1, e_2, \dots, e_n\} \tag{3}$$

These embeddings are processed by a multi-layered transformer encoder that learns contextual relationships between tokens via a self-attention mechanism, which is mentioned in Eqs. (4)–(6). The transformer encoder generates contextual representations of every token in the sequence. Lastly, a pooling operation combines token representations into a fixed-length semantic embedding that represents the entire code fragment. This semantic representation is contextual information based on vulnerability-relevant code fragments and serves as the semantic feature representation used at later stages of the identified framework.

$$h_i^{(l)} = Encoder_{\text{CodeBERT}}\left(h_i^{(l-1)}\right), h_i^{(0)} = e_i \tag{4}$$

$$Q = HW_Q, K = HW_K, V = HW_V \tag{5}$$

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{6}$$

$$H = h_1, h_2, h_3, \dots \dots, h_n \tag{7}$$

$$h_{sem} = {}^{1}/_{n} \sum_{i=1}^{n} h_i \tag{8}$$

4.3 Structural representation of smart contracts

The structural representation of smart contracts refers to transforming raw source code into structured formats that capture both syntactic and semantic information. Instead of treating the contract as plain text, it is represented using program structures such as ASTs, CFGs, and CGs [14, 27]. The AST captures the hierarchical syntactic structure of the code, while the CFG represents the flow of execution between different statements. The CG models the interactions between functions, highlighting internal and external function calls, which are crucial for detecting vulnerabilities such as RE and AC issues. This multi-level structural representation enables a comprehensive understanding of a smart contract.

4.3.1 Abstract Syntax Tree

AST is a hierarchical syntactic structure of a smart contract. Each node in AST corresponds to a language construct such as expressions, statements, or declarations. It can be formally defined as:

$$T = (V, E) \tag{9}$$

$$x_i \in R^d \tag{10}$$

where, V is the set of nodes and $E \subseteq V \times V$ denotes the parent-child relationships. Each node $v_i \in V$ is associated with a feature vector, which captures syntactic or semantic attributes. This representation enables structured analysis of program syntax.

4.3.2 Control Flow Graph

The CFG models the execution flow of a smart contract by representing all possible execution paths. It is defined as:

$$G_{cfg} = (N, E_{cfg}) \tag{11}$$

where, N is the set of basic blocks and $E_{cfg} \subseteq N \times N$ represents control flow transitions. For any edge $(n_i, n_j) \in E_{cfg}$, control may transfer from n_i to n_j. This structure helps in analyzing execution behavior and identifying logical vulnerabilities.

4.3.3 Call Graph

The CG captures the function invocation relationships within a smart contract. It is defined as:

$$G_{cg} = (F, E_{cg}) \tag{12}$$

where, F is the set of functions and $E_{cg} \subseteq F \times F$ represents calling relationships. An edge $(f_i, f_j) \in E_{cg}$ indicates that function f_i invokes function f_j. The graph can be represented using an adjacency matrix and feature matrix are given below:

$$A \in R^{|F| \times |F|} \tag{13}$$

$$X \in R^{|F| \times d} \tag{14}$$

4.4 Graph Convolutional Networks

Unlike conventional deep models that operate on Euclidean data such as images or sequences, the GCN is designed for non-Euclidean structures in which relationships between entities are expressed as edges in a graph. By aggregating information along these edges, the GCN captures both the local neighborhood of each node and, through stacked layers, the wider structural context [28].

In this study we apply three independent GCN encoders, one for each of the structural views produced by the slicing stage: the CFG, the AST, and the CG (CG). Each graph $G^{(k)}$ for $k \in \{CFG, AST, CG\}$ is described by its adjacency matrix $A^{(k)} \in R^{n_k \times n_k}$ and a node-feature matrix $X^{(k)} \in R^{n_k \times d_k}$, where n_k is the number of nodes in the pruned graph. For the CFG and AST encoders, the initial node features are learned embeddings of the syntactic node type (statement category, expression type, control construct, and so on). For the CG encoder, the features are a small set of structural descriptors of each function node-namely a root-status indicator together with the in-degree and out-degree.

Here $H^{(k,0)} = X^{(k)}$ is the initial node-feature matrix, $W^{(l)}$ is the trainable weight matrix at layer l, and $\sigma(.)$ is a non-linear activation (ReLU). Eq. (16) gives the symmetric normalization of the adjacency matrix with added self-loops: $A^{(k)} + I$ ensures that each node includes its own features alongside those of its neighbours during aggregation, and $\widetilde{D}$ is the diagonal degree matrix of $A^{(k)} + I$. After L layers, the resulting node representations $H^{(k)} \in R^{n_k \times d'}$ form the output of the GCN encoder for graph k. Each encoder propagates information through L layers using the GCN update rule:

$$H^{(k,l+1)} = \sigma((\tilde{A})^{(k)} H^{(k,l)} W^{(l)}) \tag{15}$$

$$(\tilde{A})^{(k)} = \left(\widetilde{D}\right)^{-\frac{1}{2}} (A^{(k)} + I) \left(\widetilde{D}\right)^{-\frac{1}{2}} \tag{16}$$

$$H^{(k)} = H^{(k,l)} \in R^{n_k \times d'} \tag{17}$$

The way these per-node embeddings are subsequently consumed depends on the fusion strategy. In the baseline fusion variants (concatenation and weighted fusion), each graph's node embeddings are reduced to a single graph-level vector by mean pooling, and the three resulting vectors are concatenated with the semantic representation before classification. In the proposed slice-anchored fusion variant, the CFG node embeddings $H^{(CFG)}$ are retained at the node level and combined with the corresponding CodeBERT token-span representations at the statement level, as detailed in Section 4.6; the AST and CG embeddings continue to be globally pooled in this variant.

4.4.1 Graph-level pooling

For the baseline fusion variants, and for the AST and CG branches inside the proposed slice-anchored variant of Section 4.6, a fixed-size representation of each graph is obtained by mean-pooling the GCN node embeddings:

$$h^{(k)} = \frac{1}{n_k} \sum_{i=1}^{n_k} h_i^{(k)} \tag{18}$$

where, n_k denotes the number of nodes in the pruned graph $G^{(k)}$. The proposed slice-anchored variant deliberately retains the CFG node embeddings $H^{(CFG)}$ at the node level rather than pooling them, so that each surviving statement can be aligned to its corresponding graph position during fusion.

4.4.2 Multi-graph baseline fusion

In the baseline fusion variants, the three pooled graph vectors are concatenated into a single structural representation that summarizes syntactic structure, control flow, and inter-function interactions:

$$h_{\text{graph}} = h^{(\text{AST})} \parallel h^{(\text{CFG})} \parallel h^{(\text{CG})} \tag{19}$$

where, denotes concatenation. The full concatenation ablation pairs h_{graph} with the CodeBERT semantic vector h_{sem} before classification, while the weighted-fusion ablation of Section 4.5 combines them through learned attention weights.

4.5 Weighted fusion

The CodeBERT semantic representation captures contextual information drawn from the source code, while the graph-based representation built from the AST, CFG and CG captures structural dependencies. A natural way of combining the two modalities, used as one of the baselines in our experiments, is a weighted hybrid fusion mechanism:

$$\tilde{h}_{graph} = W_g h_{graph}, \tilde{h}_{sem} = W_s h_{sem} \quad (20)$$

$$h_{fusion} = \alpha \cdot \tilde{h}_{graph} + (1 - \alpha) \cdot \tilde{h}_{sem} \quad (21)$$

$$\alpha = \frac{\exp\left(w^T \tilde{h}_{graph}\right)}{\exp\left(w^T \tilde{h}_{graph}\right) + \exp\left(w^T \tilde{h}_{sem}\right)} \quad (22)$$

$$z = W_f h_{fusion} + b_f, \hat{y} = Softmax(z) \quad (23)$$

Instead of treating the two modalities equally, the model assigns adaptive importance to each, allowing it to balance structural and contextual cues according to the input. The weight α is computed dynamically per sample, the fused vector is passed through a fully connected layer, and a Softmax produces the predicted vulnerability class. As reported in Section 5, this strategy is outperformed in our experiments by the slice-anchored fusion mechanism introduced next.

4.6 Slice-anchored fusion

Both fusion strategies described above reduce each modality to a single globally-pooled vector before combination, which discards the alignment between individual source statements and the corresponding nodes of the CFG. The proposed slice-anchored fusion mechanism preserves this alignment by performing cross-modal combination at the statement level rather than at the graph level.

Let $S = \{s_1, s_2, \ldots, s_M\}$ denote the ordered set of statements retained by the slicing stage for a given function. From the alignment map A_f produced by Algorithm 1, each statement s_i is associated with (i) a token-index span $(s_i) = \{j_a, \ldots, j_b\}$ in the CodeBERT input and (ii) a CFG node index $\nu(s_i)$. The per-statement semantic representation is obtained by mean-pooling the CodeBERT token embeddings over the span:

$$t_i = \frac{1}{|span(s_i)|} \sum_{j \in span(s_i)} e_j \quad (24)$$

where, $e_j \in R^{d_{bert}}$ is the CodeBERT embedding of token j. The corresponding per-statement structural representation is read directly from the CFG encoder output at the aligned node:

$$n_i = H^{(CFG)}_{\nu(s_i)} \quad (25)$$

The semantic vector is then projected to the GCN hidden dimensionality and fused with the structural vector through a small multi-layer perceptron ϕ:

$$h_i = \phi(W_t t_i \parallel n_i) \quad (26)$$

Eq. (26) binds the semantic content of statement s_i to its position in the CFG; this per-statement cross-modal binding is the central operation of the proposed mechanism. The fused statement vectors are aggregated into a single slice-level representation by mean-pooling:

$$z_{stmt} = \frac{1}{M} \sum_{i=1}^{M} h_i \quad (27)$$

The AST and CG views, which are not statement-aligned, continue to be summarised globally via Eq. (18). The three resulting representations are combined through learned softmax-normalised weights:

$$z = \alpha_{stmt} z_{stmt} + \alpha_{ast} h^{(AST)} + \alpha_{cg} h^{(CG)} \quad (28)$$

$$\hat{y} = Softmax\left(W_f z + b_f\right) \quad (29)$$

By performing cross-modal fusion before any global pooling step on the CFG side, the model is able to represent ordered, control-dependent patterns such as the call-before-update structure characteristic of RE-a pattern that global fusion strategies necessarily blur.

5. EXPERIMENTS

5.1 Dataset

The dataset consists of 6,698 Solidity smart contracts collected from publicly available smart-contract repositories. Vulnerability labels for the RE, access-control, and TX classes were derived from the SmartBugs Wild [29] consensus annotations, which aggregate the outputs of multiple production static analysers (including Slither, Mythril) to mitigate the limitations of any single tool, and were supplemented by manual curation for the access-control and TX classes where the automated consensus exhibits lower coverage. The non-vulnerable contracts were drawn from the same pool, with labels confirmed by the consensus analysers' agreement on the absence of the studied vulnerabilities. After compiling and slicing 6,603 of the 6,698 candidate contracts (98.6%) and deduplicating the resulting function records by normalized body hash, the working set comprised 29,347 distinct functions: 27,357 non-vulnerable, 1,025 RE, 628 access-control, and 337 TX. The dataset was first partitioned using a 70/15/15 group-aware split based on base contract identifiers, so that functions from a single contract never appear across more than one partition. Because the non-vulnerable class dominates the raw data, it was undersampled within the training partition alone, down to a negative-to-positive ratio of 1.0. This yielded a training set of 2,780 functions, comprising 1,390 non-vulnerable, 718 RE, 451 access-control, and 221 tx-origin instances. The validation set (3,886 functions: 3,646 non-vulnerable, 123 RE, 69 access-control, 48 tx-origin) and the test set (4,796 functions: 4,436 non-vulnerable, 184 RE, 108 access-control, 68 tx-origin) were left untouched, preserving the dataset's original class skew for evaluation. Since balancing relies solely on removing excess majority-class samples from training, rather than generating synthetic minority-class data, there is no mechanism by which information could leak between partitions.

5.2 Performance evaluation

This study used four evaluation measures:

$$Accuracy(ac) = \frac{tp + tn}{tp + tn + fp + fn} \quad (30)$$

$$Sensitivity(se) = \frac{tp}{tp + fn} \quad (31)$$

$$Specificity(sp) = \frac{tn}{tn + fp} \quad (32)$$

$$F1 - Score(F1) = \frac{2tp}{2tp + fp + fn} \quad (33)$$

5.3 Hyperparameter tuning

The proposed model was implemented in Python using PyTorch, PyTorch Geometric, and the Hugging Face Transformers library. Solidity source contracts were compiled with the solc compiler, with the version for each contract selected automatically from its pragma directive through solc-select; out of 6,698 candidate contracts, 6,603 compiled successfully (98.6%), and the remaining 95 were excluded from the dataset. The CFGs, ASTs, and CGs used by the structural branches were extracted using the Slither static analysis framework through its Python API, and the SVPS slicing procedure produces, for each function, both the backward slice rooted at the labelled vulnerability sinks and a character-span-to-CFG-node alignment that the fusion mechanism consumes directly. The semantic encoder is the microsoft/codebert-base variant of CodeBERT, a 12-layer transformer with 768-dimensional hidden states and a maximum input length of 256 tokens. Each of the three graph branches uses a two-layer GCN following Kipf and Welling, with 32-dimensional node-type embeddings projected to a 128-dimensional hidden space; the CFG branch returns per-node embeddings to support statement-level alignment, while the AST and call-graph branches apply mean pooling.

Table 1. Hyperparameter settings of the proposed model

Hyperparameter	Value
Learning rate	1×10^{-3}
Batch size	16
Optimiser	Adam
Training budget	25 epochs
Loss function	Class-weighted cross-entropy
CodeBERT embedding dimension	768
CodeBERT maximum sequence length	256
GCN hidden dimension	128
GCN layers per branch	2
Node-type embedding dimension (CFG, AST)	32
Dropout rate	0.3
Activation function	ReLU

Training uses the Adam optimizer with two parameter groups: the unfrozen CodeBERT parameters (the top four transformer layers and the pooler, approximately 28.9 million trainable parameters) at a learning rate of 2e-5, and the fusion head and GCN encoders (approximately 0.21 million parameters) at 1e-3. The loss function is class-weighted cross-entropy with weights proportional to the inverse square root of class frequency, normalized so the weights sum to the number of classes; the batch size is 16, dropout is 0.3, and the maximum training budget is 25 epochs with early stopping applied at patience 5 on validation macro-F1. All experiments were performed on Google Colab Pro using a single NVIDIA T4 GPU running Ubuntu 22.04, with one seed of the proposed model taking approximately 20 minutes of wall-clock time at the tuned configuration. Each configuration was trained from scratch with three or more random seeds, and reported results are means with standard deviations across those seeds. The final configuration adopted for the experimental evaluation is summarized in Table 1.

6. RESULTS AND DISCUSSION

In order to test the efficacy of the proposed approach, we compared it with conventional smart contract vulnerability detection tools, namely Slither and Mythril. These tools use rule-based and static analysis techniques. These two tools are designed to detect specific types of vulnerabilities. The comparative analysis has been performed in three categories of vulnerabilities, namely RE, AC, and TX, using the evaluation measures, which include the Accuracy, Precision, Recall, and F1-score and the results of the experiment are presented in Figure 5 and Figure 6.

Among the compared methods, the proposed method achieves significantly higher values across all evaluation metrics, indicating its strong capability in identifying RE vulnerabilities. The improvement is particularly notable in terms of precision and F1-score, reflecting more accurate and balanced detection. Within the AC vulnerabilities, traditional methodologies typically exhibit moderate efficiency, with significant variation across performance indicators. On the other hand, when it comes to vulnerabilities related to TX, the effectiveness of the implemented tools is significantly reduced and unstable. In turn, these empirical findings support the idea that the given approach yields significantly better outcomes in detecting vulnerabilities in smart contracts.

To further decompose the role of each component within the proposed framework, a sequence of experiments is performed. The first stage of the model is tested using only CodeBERT's semantic features, without any structural information. The findings suggest that although CodeBERT is an effective extractor of contextual and semantic patterns, it is nevertheless unable to capture the structural relationships inherent in smart contracts, resulting in mediocre performance across all measures. Additional experiments have been conducted using DL models such as LSTM, BiLSTM, and BERT; the results are listed in Table 2. This analysis aims to assess the consistency of CodeBERT's performance gains relative to other sequence and transformer-based architectures. The LSTM and BiLSTM have relatively lower performance, indicating limited ability to learn complex contextual dependencies in smart contract code. Transformer-based models, on the contrary, are more effective. BERT demonstrates significant progress over other recurrent models by achieving an accuracy of 87.45%, and embedding contextualization is a crucial factor. CodeBERT also improves performance across all evaluation metrics, achieving 89.46% accuracy, 87.35% sensitivity demonstrating the benefit of domain-specific pre-training for source code comprehension. These findings confirm the idea that the use of CodeBERT is

not a random decision but a rational one, as it continues to show better performance than the other baseline models. Further experiments are carried out separately using graph-based representations: AST, CFG, and CG. A graph will capture certain elements of the contract behaviour when considered individually. The AST-based model is effective at learning syntactic patterns, the CFG-based model is successful at recording the dependencies of the execution flow, and the CG is effective at examining inter-function interactions. However, in isolation, these representations offer only limited performance, insufficient to provide the full structural and semantic information needed for accurate vulnerability detection. The additional experiments are then conducted by integrating the graph representation (AST + CFG + CG), leading to performance improvements that are clearly superior to those of isolated graph-based models, the detailed results are shown in Table 3. This shows that when a multitude of structural views is incorporated, the model will be better able to comprehend a system's program behaviour.

To assess the concatenation/weighted comparison strategies are evaluated and presented in Table 4. The concatenation-based fusion method achieved an accuracy of 89.46% and an F1-score of 87.76%, and is close to the standalone CodeBERT model. It means that the feature concatenation is limited in its ability to fully exploit the complementary information available in the semantic and structural representations. The proposed slice anchored approach is, conversely, much more efficient across all evaluation metrics, achieving an accuracy of 90.48%, precision of 89.32%, recall of 89.74%, and F1-score of 89.62%. The weighted feature fusion achieved an improvement over the concatenation method.

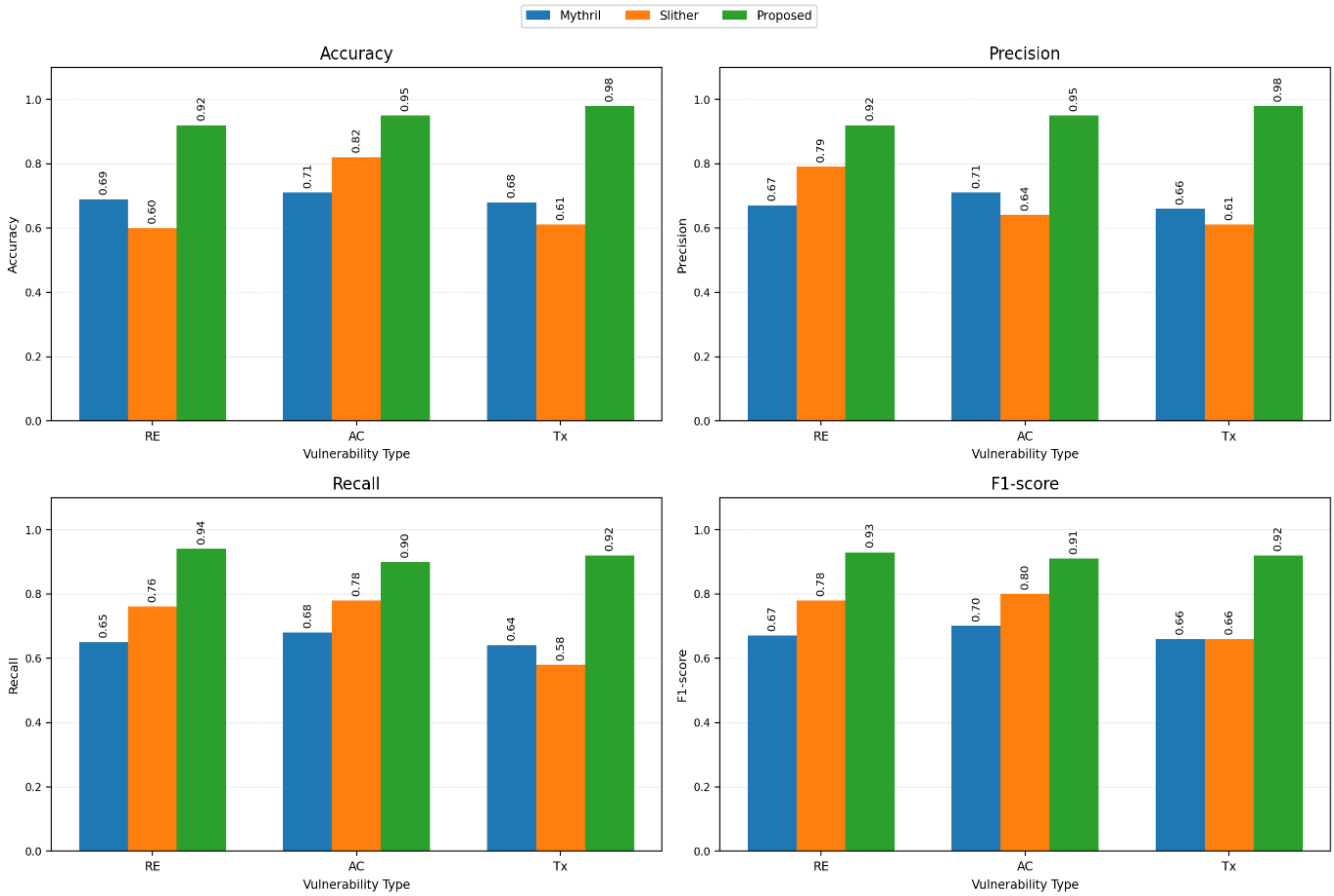

Figure 5. Performance comparison of the proposed method against baseline tools

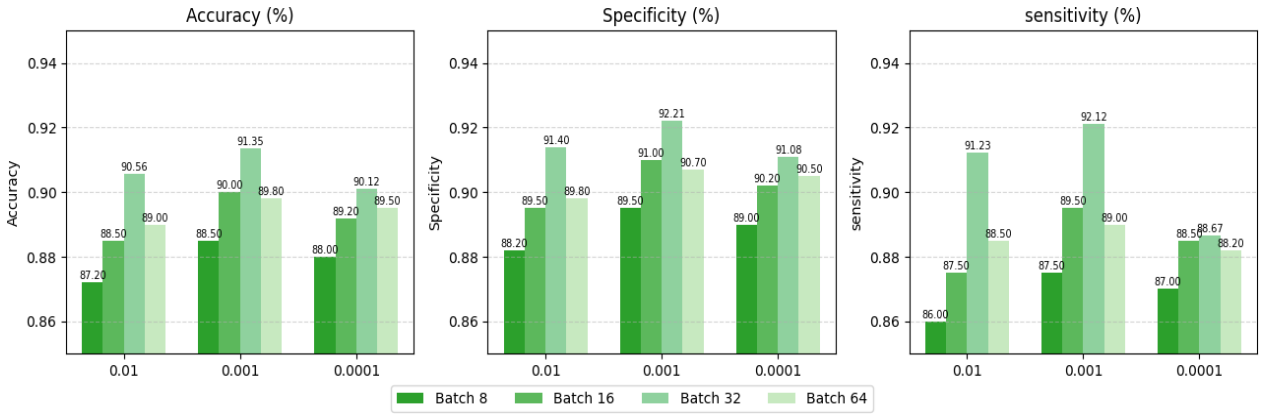

Figure 6. Effect of learning rate and batch size on model performance

Table 2. Performance comparison of semantic feature encoders

Model	Accuracy	Precision	Recall	F1-Score
LSTM	0.7712	0.7545	0.7421	0.7482
BiLSTM	0.8026	0.7883	0.7754	0.7818
BERT	0.8745	0.8612	0.8527	0.8569
CodeBERT	0.8946	0.8818	0.8735	0.8776

Table 3. Performance comparison of graph-based features

Graph Type	Accuracy	Precision	Recall	F1-Score
AST Only	0.7034	0.6812	0.6725	0.6768
CFG Only	0.7241	0.7015	0.6932	0.6973
CG Only	0.7416	0.7234	0.7128	0.7181
AST + CG	0.8043	0.7817	0.7735	0.7776
CFG + CG	0.8342	0.8135	0.8047	0.8091
AST + CFG	0.8218	0.8026	0.7942	0.7984
AST + CFG + CG	0.8719	0.8523	0.8436	0.8479

Table 4. Concatenation vs. weighted fusion vs. slice-anchored

Fusion Method	Accuracy	Precision	Recall	F1-Score
Concatenation (CodeBERT + Graph)	0.8946	0.8818	0.8735	0.8776
Weighted Fusion	0.9048	0.8932	0.8974	0.8962
Slice-Anchored Fusion (Proposed)	0.9135	0.9087	0.9212	0.9149

The proposed Slice-Anchored Fusion outperforms both baselines across every evaluation metric. It achieves an accuracy of 91.35%, a precision of 90.87%, a recall of 92.12%, and an F1-score of 91.49%. These findings indicate that the Slice-Anchored Fusion can dynamically balance the roles of semantic and structural features and provide a complete set of features. Figures 7–9 presents the confusion matrices for the codeBERT based model, graph-based model and the proposed approach, highlighting the improved class-wise prediction capability of the proposed method, and Figure 10 shows the training accuracy and validation accuracy curves for the proposed approach.

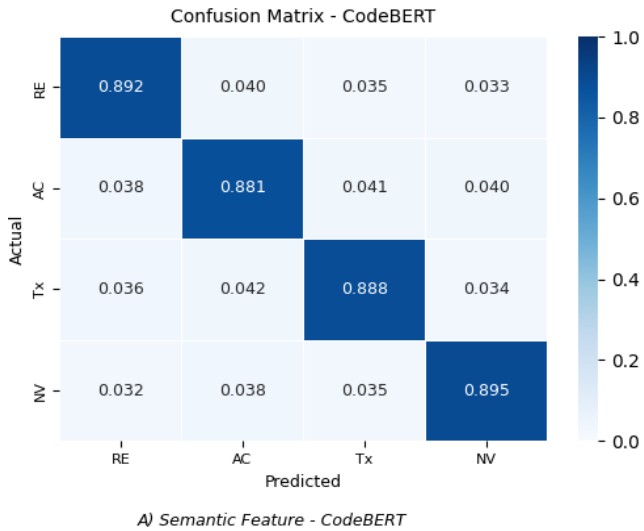

Figure 7. Confusion matrix for semantic features

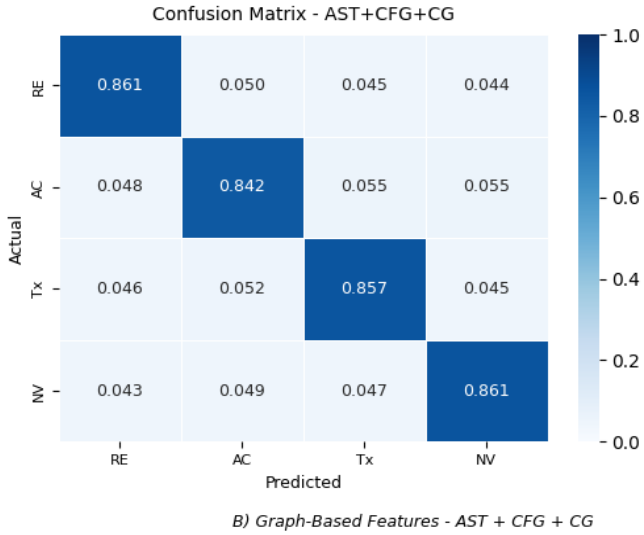

Figure 8. Confusion matrix for structural features

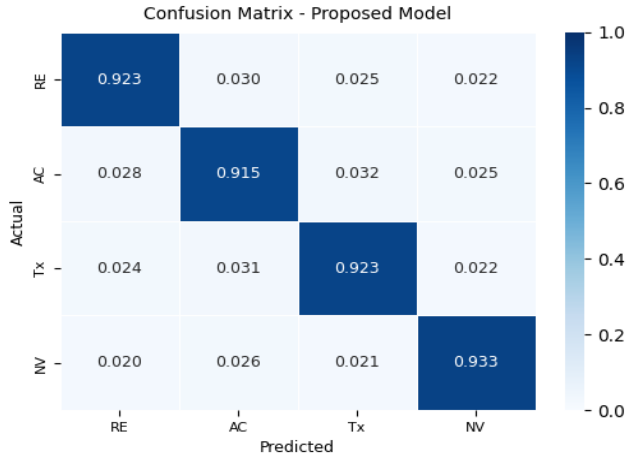

Figure 9. Confusion matrix for the proposed feature fusion model

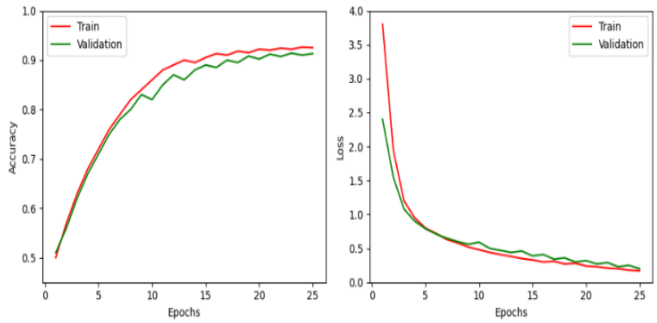

Figure 10. Training and validation accuracy and loss curves

The next experiment is conducted to analyses the importance of the proposed SVPS code slicing technique. Table 5 presents the experimental results with and without the code-slicing technique. The code-slicing model achieves 87.24% accuracy and 85.68% F1-score, indicating the presence of irrelevant and redundant information. With the addition of code slicing, accuracy improves to 91.35 and 91.49, respectively. This amounts to a 4.1% increase in accuracy and a 5.8% increase in F1-score, confirming the efficacy of targeting vulnerability-relevant code segments.

Table 5. Effect of code slicing on vulnerability detection

Model Variant	Accuracy	Precision	Recall	F1-Score
Without Code Slicing	0.8724	0.8612	0.8526	0.8568
With Code Slicing(Proposed)	0.9135	0.9087	0.9212	0.9149

Table 6. Comparison against existing baseline methods

Method	NV	AC	RE	TX	Macro-F1 ± σ	n
DeepFusion [30]	0.97	0.47	0.89	0.87	0.802 ± 0.007	3
Slice-LSTM [31]	0.98	0.44	0.91	0.86	0.795 ± 0.008	3
Peculiar [11]	0.98	0.49	0.90	0.89	0.815± 0.006	3
Cai et al. [22]	0.98	0.51	0.90	0.91	0.825 ± 0.005	3
Slice-anchored fusion	0.99	0.53	0.91	0.93	0.840± 0.005	3

Note: n independent training runs with different random seeds.

Table 6 presents baseline comparison across every vulnerability category. AC remains the hardest class throughout, with F1-scores rising from 0.44 (Slice-LSTM) to 0.53 (proposed). Non-vulnerable contracts are detected reliably by all methods, with F1-scores above 0.97. Macro-F1 improves steadily from 0.795 to 0.840, a 1.5-point gain over Cai et al. [22], the strongest baseline. The proposed method also matches the lowest run-to-run variability (σ = 0.005), showing the improvement is stable, not incidental.

7. CONCLUSIONS

Smart-contract vulnerabilities continue to threaten blockchain-based systems, and reliable detection remains a difficult problem because the structural patterns that distinguish vulnerable code from safe code are easily lost when contracts are presented to neural models as flat token sequences. In this work we addressed that gap with a function-level multimodal framework that detects vulnerabilities by combining the semantic content and the structural behavior of smart contracts at the granularity of individual statements. Rather than passing entire contracts through the model, each function is first reduced by SVPS, a sink-driven backward-slicing procedure that retains only the statements connected to vulnerability triggering operations through data or control dependence. The slice is then encoded along two parallel paths: CodeBERT produces a contextual semantic representation of the surviving source tokens, while three GCN encoders process the pruned CFG, AST, and CG to capture syntactic, control flow, and inter-function structure. Instead of merging these two views as globally-pooled vectors, as prior multimodal detectors do, the proposed slice-anchored fusion mechanism aligns each sliced statement's CodeBERT token span with its corresponding CFG node and fuses them at the statement level, preserving the code-to-graph correspondence that global pooling discards.

REFERENCES

[1] Qian, P., Liu, Z.G., He, Q.M., Huang, B.T., Tian, D.Z., Wang, X. (2022). Smart contract vulnerability detection technique: A survey. arXiv preprint, arXiv: 2209.05872. https://doi.org/10.48550/arXiv.2209.05872

[2] Zhang, L.J., Wang, J.L., Wang, W.Z., Jin, Z.L., Su, Y.S., Chen, H.L. (2022). Smart contract vulnerability detection combined with multi-objective detection. Computer Networks, 217: 109289. https://doi.org/10.1016/j.comnet.2022.109289

[3] Zou, W.Q., Lo, D., Kochhar, P.S., et al. (2019). Smart contract development: Challenges and opportunities. IEEE Transactions on Software Engineering, 47(10): 2084-2106. https://doi.org/10.1109/TSE.2019.2942301

[4] Cai, J., Li, B., Zhang, T., Zhang, J.L., Sun, X.B. (2024). Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction. Journal of Systems and Software, 209: 111919. https://doi.org/10.1016/j.jss.2023.111919

[5] Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I. (2018). A survey of symbolic execution techniques. ACM Computing Surveys, 51(3): 50. https://doi.org/10.1145/3182657

[6] Wu, H.J., Zhang, Z., Wang, S.W., et al. (2021). Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), Wuhan, China, pp. 378-389. https://doi.org/10.1109/ISSRE52982.2021.00047

[7] Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A., Cha, S.K. (2021). Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, pp. 227-239. https://doi.org/10.1109/ASE51524.2021.9678888

[8] Sharma, N., Sharma, S. (2022). A survey of Mythril, a smart contract security analysis tool for EVM bytecode. Indian Journal of Natural Sciences, 13(75): 51003-51010.

[9] Feist, J., Grieco, G., Groce, A. (2019). Slither: A static analysis framework for smart contracts. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, pp. 8-15. https://doi.org/10.1109/WETSEB.2019.00008

[10] Xu, Y.J., Hu, G.R., You, L., Cao, C.T. (2021). A novel machine learning-based analysis model for smart contract vulnerability. Security and Communication Networks, 2021: 5798033. https://doi.org/10.1155/2021/5798033

[11] Wu, H., Zhang, Z., Wang, S., et al. (2021). Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), Wuhan, China.

[12] Liu, Y., Wang, C., Ma, Y. (2024). DL4SC: A novel deep

learning-based vulnerability detection framework for smart contracts. Automated Software Engineering, 31(1): 24. https://doi.org/10.1007/s10515-024-00418-z

[13] Xiong, Z.G., Lou, Q.Q., Li, Y.F., Chen, H., Zhang, X.M., Li, Y., Li, J. (2025). NDLSC: A new deep learning-based approach to smart contract vulnerability detection. Journal of Signal Processing Systems, 97(1): 49-68. https://doi.org/10.1007/s11265-025-01954-x

[14] Pasqua, M., Benini, A., Contro, F., Crosara, M., Dalla Preda, M., Ceccato, M. (2023). Enhancing Ethereum smart-contracts static analysis by computing a precise control-flow graph of Ethereum bytecode. Journal of Systems and Software, 200: 111653. https://doi.org/10.1016/j.jss.2023.111653

[15] Liu, H., Liu, C., Zhao, W.Q., Jiang, Y., Sun, J.G. (2018). S-gram: Towards semantic-aware security auditing for Ethereum smart contracts. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18). Association for Computing Machinery, New York, NY, USA, pp. 814-819. https://doi.org/10.1145/3238147.3240728

[16] Wang, W., Song, J.J., Xu, G.Q., Li, Y.D., Wang, H., Su, C.H. (2021). ContractWard: Automated vulnerability detection models for Ethereum smart contracts. IEEE Transactions on Network Science and Engineering, 8(2): 1133-1144. https://doi.org/10.1109/TNSE.2020.2968505

[17] Liao, J.W., Tsai, T.T., He, C.K., Tien, C.W. (2019). SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, pp. 458-465. https://doi.org/10.1109/IOTSMS48152.2019.8939256

[18] Deng, W.C., Wei, H.C., Huang, T., Cao, C., Peng, Y., Hu, X. (2023). Smart contract vulnerability detection based on deep learning and multimodal decision fusion. Sensors, 23(16): 7246. https://doi.org/10.3390/s23167246

[19] Zhuang, Y., Liu, Z.G., Qian, P., Liu, Q., Wang, X., He, Q.M. (2020). Smart contract vulnerability detection using graph neural networks. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI), pp. 3283-3290. https://doi.org/10.24963/ijcai.2020/454

[20] Liu, Z.G., Qian, P., Wang, X.Y., Zhuang, Y., Qiu, L., Wang, X. (2023). Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Transactions on Knowledge and Data Engineering, 35(2): 1296-1310. https://doi.org/10.1109/TKDE.2021.3095196

[21] Nguyen, H.H., Nguyen, N.M., Xie, C., Ahmadi, Z., Kudendo, D., Doan, T.N., Jiang, L. (2022). MANDO: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities. In 2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA), Shenzhen, China, pp. 1-10. https://doi.org/10.1109/DSAA54385.2022.10032337.

[22] Cai, J., Li, B., Zhang, J.L., Sun, X.B., Chen, B. (2023). Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. Journal of Systems and Software, 195: 111550. https://doi.org/10.1016/j.jss.2022.111550

[23] Ma, C., Liu, S.W., Xu, G.X. (2023). HGAT: Smart contract vulnerability detection method based on hierarchical graph attention network. Journal of Cloud Computing, 12(1): 93. https://doi.org/10.1186/s13677-023-00459-x

[24] Sh[ang, J.D., Li, J.R., Sui, Y.Z., Guo, H.L., Gao, X., Zhang, D.J., Guo, Y., Wu, G. (2025). CEGT: Smart contract vulnerability detection via connectivity-enhanced GCN-transformer. Journal of Systems and Software, 227: 112454. https://doi.org/10.1016/j.jss.2025.112454

[25] Duy, P.T., Khoa, N.H., Quyen, N.H., et al. (2025). VulnSense: Efficient vulnerability detection in Ethereum smart contracts by multimodal learning with graph neural network and language model. International Journal of Information Security, 24: 48. https://doi.org/10.1007/s10207-024-00965-2

[26] Feng, Z.Y., Guo, D.Y., Tang, D.Y., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 1536-1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[27] Li, Z., Zou, D.Q., Xu, S.H., et al. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA, pp. 1-15. https://doi.org/10.14722/ndss.2018.23158

[28] Kipf, T.N., Welling, M. (2017). Semi-supervised classification with graph convolutional networks. arXiv preprint, arXiv: 1609.02907. https://doi.org/10.48550/arXiv.1609.02907

[29] Ferreira, J.F., Cruz, P., Durieux, T., Abreu, R. (2020). SmartBugs: A framework to analyze Solidity smart contracts. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020), Melbourne, Australia, pp. 1349-1352. https://doi.org/10.1145/3324884.3415298

[30] Chu, H., Zhang, P., Dong, H., Xiao, Y., Ji, S. (2025). DeepFusion: Smart contract vulnerability detection via deep learning and data fusion. IEEE Transactions on Reliability, 74(3): 3544-3558. https://doi.org/10.1109/TR.2024.3480010

[31] Yu, X., Zhao, H., Hou, B., Ying, Z., Wu, B. (2021). DeeSCVHunter: A deep learning-based framework for smart contract vulnerability detection. In 2021 International Joint Conference on Neural Networks (IJCNN), Shenzhen, China. https://doi.org/10.1109/IJCNN52387.2021.9534324