Journal of Artificial Intelligence and Technology, 2026, 6, 804-814
https://doi.org/10.37965/jait.2026.1223

ISTP
RESEARCH ARTICLE

TCRNet: Topology-Guided Contextual Representation Network for Smart Contract Vulnerability Detection

S. Rajeshwari[1,2] and Mahantesh Mathapati[1]

[1]Department of Computer Science and Engineering, Amruta Institute of Engineering and Management Sciences, Bidadi-562109, Affiliated to Visvesvaraya Technological University, Belagavi-590018, India

[2]Department of Information Science and Engineering, SJB Institute of Technology, Bengaluru-560060, India

(Received 23 February 2026; Revised 24 July 2026; Accepted 01 August 2026; Published online 03 September 2026)

Abstract: Smart contracts operate in decentralized environments where deployed code cannot be easily modified, making security vulnerabilities particularly critical. Even minor logical flaws may lead to severe financial and operational consequences. Although traditional static analysis and symbolic execution techniques have been widely used for vulnerability detection, they often rely on handcrafted rules and struggle to generalize to complex or evolving contract patterns. Recent deep learning approaches have improved detection performance by learning representations directly from code, yet many of these models treat source code primarily as token sequences or graph structures in isolation, limiting their ability to capture deeper semantic interactions. We propose Topology-Guided Contextual Representation Network (TCRNet), a Contextual Encoding Network framework for smart contract vulnerability detection. The model integrates multi-granular semantic information and structural dependency knowledge directly into the attention mechanism. Experiments conducted on a large-scale dataset derived from SmartBugs demonstrate that the proposed approach significantly outperforms traditional static tools and recent learning-based methods. TCRNet achieves an accuracy of 90.14% and a macro F1-score of 94.67%, showing substantial improvements over strong baselines. These results indicate that integrating semantic richness and structural awareness within a Contextual Encoding Network architecture enhances both detection reliability and generalization capability, making the framework suitable for practical smart contract security analysis.

Keywords: Contextual Encoding Network; control-flow and data-flow analysis; smart contract vulnerability detection; semantic-enriched code representation; topology-guided attention mechanism

I. INTRODUCTION

Blockchain technologies have fundamentally transformed how trust is established in distributed networks, and smart contracts are central to this change. They are programs that run on a blockchain and automatically enforce agreements. Smart contracts have enabled a wide range of applications, from decentralized finance and asset management to voting systems and supply-chain tracking [1,2], by removing intermediaries and making execution deterministic. However, as their adoption has grown, concerns regarding their security and reliability have grown as well [3]. Unlike conventional software, smart contracts operate in an immutable and adversarial environment. Once deployed, their code cannot be easily modified, and any flaw in the logic may be exploited indefinitely. Over the past decade, several real-world incidents have shown that flaws in smart contracts can cause substantial financial losses and undermine trust in blockchain systems. Such weaknesses often originate from subtle issues in control flow, data dependencies, or improper handling of external calls, which are difficult to detect through manual inspection of the code [4].

A variety of automated vulnerability detection methods have been investigated to address these problems. Traditionally, most methods relied on static analysis, symbolic execution, and formal verification. These methods are effective in certain settings, but they generally depend on handcrafted rules or rigid predefined specifications, and they often struggle with complex contracts or programming patterns that evolve over time. These limitations have become more apparent as smart contract ecosystems have grown [5]. In parallel, advances in machine learning have made automated code analysis increasingly feasible. Learning-based solutions reduce the reliance on expert-defined rules by learning patterns directly from data [4], and they can accommodate diverse coding styles. Early efforts in this direction treated code as a sequence of tokens and adopted techniques from natural language processing (NLP). Subsequent studies employed syntactic structures such as abstract syntax trees to better represent hierarchical relationships. More recently, graph-based representations have gained popularity because they naturally capture semantic relations such as control flow and data dependencies, which are essential for understanding program behavior [5].

This line of research has advanced further through Contextual Encoding Network architectures. They are well suited to analyzing complex programs in which vulnerability-related statements may be widely separated in the source code, since they can model long-range dependencies. However, source code is not merely a sequence of symbols; its meaning is closely tied to its execution behavior and the rules governing its structure. Consequently, effective vulnerability identification requires models that move

Corresponding author: Ms. S. Rajeshwari (e-mail: rajeshwaris1992@rediffmail.com).

beyond surface-level context and account for the semantics of the program [6]. Recent architectural designs therefore combine program analysis methodologies with Transformer-based models, incorporating structural and semantic information directly into token representations and attention processes. Such techniques aim to produce representations that more faithfully reflect program behavior by aligning learning objectives with the underlying execution logic of the code [7]. Within this broader context, semantic-aware Transformer architectures provide a solid foundation for automatically detecting vulnerabilities in smart contracts, combining the flexibility of deep learning with the rigor of program semantics.

Smart contract vulnerabilities frequently stem from intricate interconnections among control flow, data dependencies, and execution semantics, rather than from discrete code segments. Accurately identifying such weaknesses requires models that can reason about both local statements and long-range dependencies in contract code. Many existing automatic detection methods, however, either rely on surface-level token representations or capture only a limited and indirect view of program semantics [8]. Graph-based representations enhance semantic modeling by capturing control and data relationships; yet, the successful integration of this information into learning systems continues to pose challenges. Transformer-based models are effective at capturing long-range correlations, but when applied directly to source code, they lack awareness of the structure and logic of the program. Because of this, attention mechanisms might not focus on code areas that are semantically important for finding vulnerabilities [9]. The central problem addressed in this work is how to build a vulnerability detection framework that explicitly incorporates program semantics, such as dependency structure, token roles, and structural relevance, into Transformer-based learning. Such a framework must support accurate reasoning over smart contract code while remaining able to handle complex dependency patterns and limited labeled data [10].

A. MOTIVATION AND CONTRIBUTION

Smart contracts are deployed in decentralized environments where code, once published, cannot be easily modified. Any vulnerability, even a small logical oversight, may lead to financial loss or permanent system failure. Although several static analysis tools and symbolic execution methods have been proposed, many of them rely heavily on predefined rules or pattern matching. These approaches often fail when vulnerabilities arise from complex interactions within the program logic rather than obvious syntactic flaws.

Recent deep learning models have attempted to improve detection by learning patterns directly from source code. However, many existing models focus primarily on token sequences or graph structures in isolation. In practice, smart contract vulnerabilities frequently emerge from subtle relationships between control flow, data dependencies, and execution context. Capturing these interactions requires more than surface-level representation. Models that ignore deeper semantic roles or structural dependencies may achieve reasonable performance on benchmark datasets but struggle to generalize across diverse contract styles and real-world scenarios [11].

Transformer-based architectures offer strong capability in modeling long-range dependencies, yet when applied directly to code, they lack inherent understanding of program structure. Source code is not merely text; it carries execution semantics and structured relationships that must be explicitly modeled. This observation motivates the need for a framework that integrates structural and semantic knowledge directly into the learning process rather than treating them as secondary inputs. Therefore, the motivation of this work is to design a vulnerability detection model that is structurally aware and semantically informed. By embedding dependency relationships, contextual interactions, and execution-relevant features into the attention mechanism, the proposed approach seeks to improve detection reliability while reducing false positives. The ultimate goal is to develop a more robust and generalizable framework for smart contract security analysis that remains computationally efficient and practical for real-world deployment.

The main contributions of this work are summarized as follows:

(1) **Topology-Guided Contextual Representation Network (TCRNet).** We propose a structurally aware smart contract vulnerability detection (SCVD) framework that integrates semantic dependencies and program structure directly into the learning process [12]. Unlike conventional token-based or graph-only models, the proposed approach captures control flow, data dependencies, and contextual relationships in a unified representation to improve detection reliability.

(2) **Structure-Guided Attention Mechanism.** A novel attention mechanism is introduced that embeds dependency relationships, shortest-path information, and semantic roles into the model's attention computation. By guiding attention toward security-critical code regions, the framework enhances vulnerability localization while reducing false positives and improving generalization across diverse contract patterns.

(3) **Improved Robustness and Practical Efficiency.** Extensive experiments demonstrate that the proposed model achieves superior detection performance compared to existing static analysis and deep learning approaches, while maintaining computational efficiency suitable for real-world smart contract auditing environments.

II. RELATED WORK

Traditional SCVD technologies mostly used static and dynamic analysis methods, such as symbolic execution and formal verification, to find any security holes. Oyente [6] was a tool that used symbolic execution to look at smart contract bytecode. It used control-flow graphs and ultimately became the basis for more complex detectors like Maian [7]. Despite its wide adoption, symbolic execution is computationally expensive and often doesn't cover all data flows, which can mean that vulnerabilities are ignored. To solve the problem of symbolic execution-based methods taking a long time, Zheng *et al.* [8] came up with Park, which used parallel execution and several CPU cores to speed up analysis. Slither [9] created a framework for static analysis that used software analysis methods like taint tracking and data-flow analysis to find weaknesses in smart contracts. Mythril [10] used symbolic execution, taint analysis, and control-flow verification to find common patterns of vulnerabilities. Defectchecker [11] also employed symbolic execution and looked for three important things—currency calls, block loops, and fallback functions—to find eight different sorts of vulnerabilities. These conventional SCVD approaches worked well in some situations, but they relied a lot on rules set by experts, which made them easy for sophisticated attackers to circumvent. Moreover, real-world attacks often combine several vulnerabilities, which makes detection even more difficult and reduces the effectiveness of rule-based automation.

There are two main types of deep learning-based SCVD methods: those that use NLP and those that use graphs. NLP-based methods look at the source code or bytecode of smart contracts as sequences of tokens and see finding vulnerabilities as a text classification problem. ContractWard [12] uses 2-gram features, but this causes a lot of extra features to be included. Duan *et al.* [13] improve detection by combining features from the opcode level and the source level. ESCORT [14] uses embedding techniques and gated recurrent units (GRUs) to find semantic and temporal relationships. It also uses transfer learning to deal with new vulnerabilities that come up. Clear [15] uses contrastive learning to find fine-grained semantic connections, which makes it easier to find vulnerabilities. Overall, NLP-based approaches work well because they can build features in a flexible way and train from start to finish, although they could miss deeper structural information.

Graph-based methods get around this problem by turning smart contracts into non-Euclidean structural representations, including control-flow graphs or abstract syntax trees, and then using graph neural networks to learn graph embeddings [16]. Zhuang *et al.* [17] put forward DGGCN and Temporal Message Propagation (TMP) for classifying graphs, while Bytecode and Source-code based Graph Vulnerability Detection (BSGVD) [18] combines abstract syntax trees and control-flow graphs to capture both syntactic and execution semantics at the same time. Jiang *et al.* [19] use community-aware features and multi-task learning to enhance detection accuracy. Dynamic execution has also been looked into for finding vulnerabilities, in addition to static code analysis. TxSpector [20] replays past transactions, records execution traces at the bytecode level, and extracts control-flow and data-flow dependencies, which are then turned into logical relations for user-defined queries. TxSpector is different from predictive, code-centric deep learning methods since it focuses on dynamic, post hoc forensic investigation of real-world attacks. Despite their effectiveness, these graph-based approaches share recurring limitations: they generally rely on a single structural view, propagate edge information only locally through message passing, and rarely couple fine-grained token-level semantics with global dependency structure, which constrains their ability to model the long-range, cross-statement interactions that underlie many smart contract vulnerabilities.

A parallel line of work adapts pre-trained Transformer models of source code to vulnerability detection. CodeBERT [21] learns bimodal representations of natural language and programming language but treats code as a flat token sequence. GraphCodeBERT [22] augments pre-training with data-flow information, while UniXcoder [23] unifies encoder–decoder objectives and incorporates abstract syntax trees and code comments through generic cross-modal pre-training. More recently, 2024–2025 studies have explored large language model (LLM)-based auditing, such as iAudit [24], which combines fine-tuning with LLM agents to produce vulnerability justifications, and multi-modal detectors that fuse source code with bytecode, including cross-modality mutual learning [25] and multi-view graph contrastive learning [26]. Unlike these models, which rely primarily on generic pre-training objectives or treat structure as an auxiliary signal, the proposed TCRNet injects explicit dependency-type and shortest-path biases directly into the attention computation at the detection stage, coupling multi-granular semantic embeddings with task-specific structural guidance. This design targets the precise control- and data-dependency interactions that characterize smart contract vulnerabilities, rather than relying on representations learned for general-purpose code understanding.

III. PROPOSED MODEL

Detecting vulnerabilities in source code is a critical task for ensuring software security, yet it remains challenging due to the complex semantics and structural dependencies present in real-world programs. Traditional static analysis tools rely heavily on handcrafted rules and predefined patterns, which limits their ability to generalize to unseen or complex vulnerability types [27]. While recent deep learning-based approaches have improved detection performance by treating source code as a sequence of tokens, many of these methods fail to fully capture the underlying semantic meaning and execution behavior of programs.

A program function is composed of multiple statements whose behavior is determined not only by their lexical elements but also by their structural relationships within the code [28]. As proposed in this paper, a buffer overflow example highlights this complexity: the conditional statement at line six conveys semantics at the line stage by defining a control branch, while simultaneously embedding token-level information such as variable references, function invocations, and operators. Accurately identifying the vulnerability further depends on understanding structural interactions, including how data produced at line five influences the conditional expression and how this condition controls the execution of subsequent statements [29]. Existing detection approaches often struggle to model these intertwined lexical, semantic, and structural signals in a unified manner, limiting their effectiveness in accurately detecting vulnerabilities.

To overcome the aforementioned limitations, we introduce TCRNet a Contextual Encoding Network-based model augmented with two modules designed to better capture code semantics and structural dependencies that are mentioned below: Semantic-Enriched Code Expression: This module enriches code embeddings by jointly modeling token-level information, such as variable and function types, statement-level semantics that describe the role of each line, and structural indicators reflecting node significance derived from program graphs. This multi-granular encoding preserves critical semantic details and highlights features that are strongly associated with vulnerable behavior during the learning process [30]. Structural Aided Dependency Modelling: To effectively integrate program structure, we construct a graph that integrates Code Dependency which consolidates both control-flow and data-flow relationships. From this unified graph, we derive matrices capturing dependency categories and relative distances, which are injected into the Contextual Encoding Network's attention mechanism as structural biases [31]. This design steers the model's attention toward security-sensitive code regions, thereby improving its ability to detect vulnerability-related dependencies.

The Semantic-Enriched embedding module maintains multi-level semantic information within code representations, whereas the dependency-oriented encoding mechanism captures structural relationships among program elements. By jointly modeling semantic richness and structural context within a unified architecture, the proposed approach strengthens the model's ability to identify context-dependent vulnerabilities that emerge from intricate interactions between code meaning and dependency patterns [32].

Fig. 1. illustrates the complete architecture of the proposed TCRNet model, which is specifically tailored for identifying vulnerabilities in the source code. In this architecture, vulnerability detection is formulated as a function-level multi-label classification task: each code function is associated with zero, one, or several vulnerability categories (arithmetic, reentrancy, timestamp

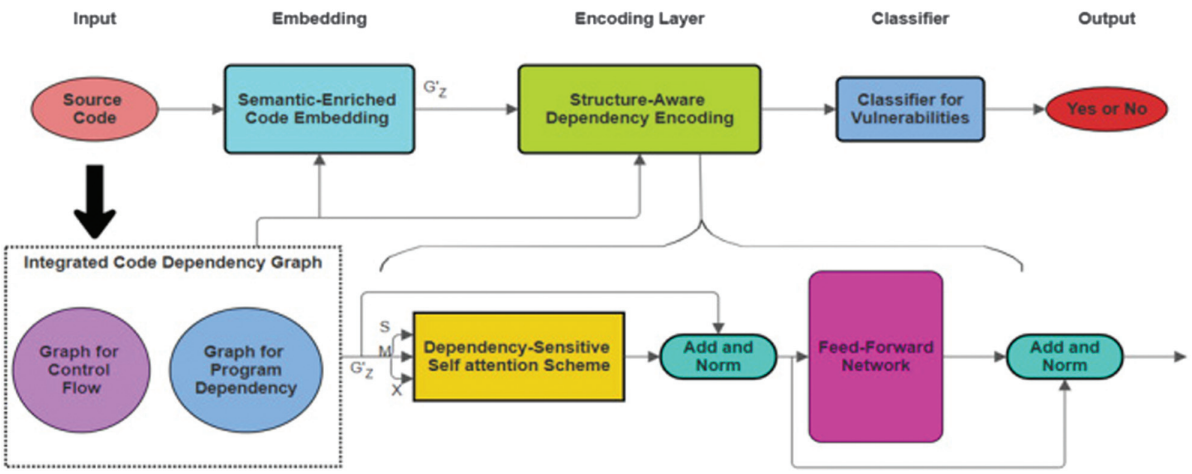

Fig. 1. Proposed model.

dependency, and unchecked low-level call), and functions without any of these flaws are assigned the clean label. Function-level labels follow the convention adopted in prior work [27]. This multi-label formulation is consistent with the labeled dataset described in Section IV.A, in which a subset of functions exhibit co-occurring vulnerabilities, and with the macro-averaged evaluation metrics reported in Section IV.C. The key contribution of this approach is the seamless incorporation of multiple forms of semantic and structural code data into the framework of the Contextual Encoding Network through two principal modules: Semantic-Enriched Code Expression model and Structural aided Dependency modeling.

At the embedding stage, TCRNet constructs semantically informed representations by combining two-dimensional types of embeddings: positional embeddings and degree-based embeddings. This design allows the model to capture the relative importance of individual statements while accounting for their roles within the broader program structure [33]. Within the encoder, spatial dependency modeling is strengthened by augmenting the self-attention mechanism with a dependency-type matrix and a newly designed dependency-aware distance matrix, enabling the model to better emphasize structurally significant relationships in the code. The encoder outputs are then used to form a unified representation of the input function by aggregating information across multiple granularities, including token-level features, statement-level semantics, and dependency-aware encodings. This consolidated, context-enriched embedding is passed to a Multi-Layer Perceptron (MLP) classifier, which serves as a completely linked prediction layer to estimate the likelihood that the given code contains vulnerabilities considering the security [34]. By jointly leveraging semantic richness and structural awareness, TCRNet achieves more accurate and reliable vulnerability detection compared to existing approaches.

In vulnerability detection tasks, accurately assessing the relevance of individual code statements is essential, which motivates the incorporation of semantic-aware representations. By

Algorithm 1. Construction of the Integrated Code Dependency Graph (ICDG)
Input: Solidity function source code F Output: ICDG G = (V, E) with typed edges {EH, EF, FF}

1: Parse F into an abstract syntax tree (AST)
2: Extract the set of statements S and their constituent tokens T from the AST
3: Build the control-flow graph CFG by linking each statement in S to its successors
4: Derive control-dependency edges E_EH from the CFG via post-dominator analysis
5: Construct data-dependency edges E_FF by def-use analysis over program variables
6: Set control-flow edges E_EF from the CFG successor relation
7: V ⊠ S, with each node aligned to its tokens in T; E ⊠ { }
8: for each ordered node pair (v_k, v_l) in V × V do
9: if control-flow relation holds then E ⊠ E ∪ {(v_k, v_l, EF)}
10: if control-dependency relation holds then E ⊠ E ∪ {(v_k, v_l, EH)}
11: if data-dependency relation holds then E ⊠ E ∪ {(v_k, v_l, FF)}
12: end for
13: Discard purely syntactic AST edges (already captured by token-level embeddings)
14: Compute in-degree and out-degree of every node (degree-based embeddings)
15: Compute all-pairs shortest distances over G (distance matrix of Eq. (7))
16: return G = (V, E)

embedding semantic information directly into the Contextual Encoding Network architecture, the softmax-based attention mechanism is better equipped to identify and prioritize critical code lines when computing query–key interactions [35]. This design enables the model to simultaneously capture semantic relationships and the relative importance of nodes during attention computation, therefore strengthening its ability to identify vulnerable patterns. Formally, the semantic-aware embeddings G'_Z are constructed for a given token sequence Z which is formulated as given in Eq. (1):

$$G'_Z = G_Z + G_V + G_R + G_F \tag{1}$$

In this case, G_Z is used to denote the textual embeddings, G_F is used to show the degree-based embeddings, G_R expresses the positional embeddings, and G_V represents the type-based embeddings.

While we consider the *type-based embeddings G_V in two-dimension*, Program statements exhibit diverse functional roles—such as assignments, control structures, and function invocations—which influence program execution in different ways. As a result, certain statement categories are more prone to introducing or propagating vulnerabilities. For example, improperly constructed loops may result in security issues such as buffer overflows or unintended infinite execution. Likewise, tokens within a statement, including variables, constants, and operators, do not contribute equally to program behavior. Arithmetic operators, in particular, can be associated with integer overflow vulnerabilities when applied without proper safeguards.

To capture these distinctions, TCRNet employs a two-dimensional type embedding strategy derived from the classification of both code statements and individual tokens. Under this scheme, every token is associated with two continuous embedding vectors: one encoding the semantic category of the statement in which the token appears, and the other representing the token's own syntactic or functional type. This dual-type encoding supplies the Contextual Encoding Network with enriched semantic cues, thereby improving the quality of source code representations. For Eq. (2) given above, $G_Z^{statement}$ and G_Z^{token} *belong to* $\mathbb{T}^{p\times f_{model}}$ that can be learnt vectors that are embedded into the code line type as well as the token, respectively:

$$G_V = G_Z^{statement} + G_Z^{token} \tag{2}$$

For the *Position-based embeddings G_R in two-dimension*, accurately modeling the order of operations and function invocations is essential for analyzing data flow, as these factors directly influence vulnerabilities arising from improper data usage, such as injection flaws as well as buffer overflows. Positional information can strengthen this understanding by enabling the model to better capture the structural organization and execution flow of source code. For example, the scope and lifetime of program elements—particularly variables—depend on where they are defined, such as within iterative blocks versus at the global level. Incorporating positional context therefore helps expose issues like the use of uninitialized variables or references that occur outside their valid scope.

To support this capability, TCRNet introduces a two-dimensional absolute positional embedding scheme that operates at both the statement and token levels. Under this design, each token is associated with two continuous embedding vectors: one encoding the position of its enclosing code line and the other representing its position within that line. These embeddings provide the Contextual Encoding Network with additional structural cues, therefore improving the expressiveness of the learned code representations. For Eq. (3), $G_R^{statement}$ and G_R^{token} *belong to* $\mathbb{T}^{p\times f_{model}}$ that can be learnt vectors that are embedded into the code statement line position as well as the token position, respectively.

$$G_R = G_R^{statement} + G_R^{token} \tag{3}$$

Considering the *Degree-based embeddings G_F in two-dimension*, Graph-based representations extracted from source code provide valuable structural and semantic context that supports a deeper understanding of program behavior. In particular, node in-degree as well as out-degree measures convey the relative importance of code elements and their interactions within the graph. For example, in a Control Flow Graph, nodes with a large number of outgoing edges often correspond to branching or decision points that significantly influence execution paths. Similarly, the Integrated Code Dependency Graph (ICDG) captures how control as well as data dependencies propagate throughout the program, with degree information indicating where such dependencies converge or branch.

In TCRNet, this degree-related information is incorporated into the Contextual Encoding Network through two-dimensional degree embeddings that encode both incoming and outgoing connections derived from the Integrated Code Dependency Graph. By injecting these structural indicators into the model, the Contextual Encoding Network is better equipped to model relational patterns within the code, which in turn enhances its effectiveness in identifying and reasoning about potential security vulnerabilities. Here, in Eq. (4), $G_F^{in\text{-}degree}$ and $G_F^{out\text{-}degree}$ *belong to* $\mathbb{T}^{p\times f_{model}}$ that can be learnt vectors that are embedded into the in-degree as well as the out-degree, respectively:

$$G_F = G_F^{in\text{-}degree} + G_F^{out\text{-}degree} \tag{4}$$

Structure-Aware Dependency Encoding: One of the strengths of the Contextual Encoding Network architecture is its global receptive domain, which allows every token to selectively integrate information from any other position in the input sequence. This global receptive capability makes Contextual Encoding Networks well suited for modeling long-range and complex relationships in source code. At the same time, such flexibility requires the model to explicitly encode positional and relational information in order to distinguish meaningful structural dependencies. To effectively exploit the dependency knowledge embedded within the Integrated Code Dependency Graph, we introduce a structure-aware spatial dependency encoding strategy within the encoder layer.

When constructing the Integrated Code Dependency Graph, we focus on three fundamental edge categories—controlling dependency, controlling flow, and data dependency—as these relationships are closely tied to common vulnerability patterns and are essential for capturing program behavior. This selective design balances semantic expressiveness with computational efficiency, ensuring that only vulnerability-relevant dependencies are modeled. A deliberate design choice is made to exclude syntactic edges derived from the Abstract Syntax Tree. Fine-tuned syntactic data characteristics are already efficiently grasped through token-stage semantic expressions, such as type-based and positional-based embeddings. Incorporating Abstract Syntax Tree edges shall substantially make the graph complexity more and computational overhead cost without providing additional vulnerability-specific insights and could potentially distract the model from more critical control- and data-oriented dependencies.

Dependency-Sensitive encoding model: The characteristics of edges in the ICDG plays an essential role in modeling relationships among program elements. While edge attributes can be aggregated alongside node features during message passing, such localized propagation limits the influence of edge information to directly connected nodes and may not sufficiently reflect global structural

patterns. To more effectively integrate edge-related semantics into the Contextual Encoding Network's attention mechanism, we propose a dependency-sensitive encoding strategy that represents multiple edge categories through a summation formulation technique, enabling both individual and composite dependency types to be flexibly modeled.

Let $G_{edge\text{-}type}$ denote the set of all edge categories in the graph. Every edge type v_k *belongs to* $G_{edge\text{-}type}$ is assigned a distinct encoding value ϑ_{v_k} *belongs to* $\mathbb{P}$. Given that the ICDG consists of three dependency types—controlling dependency, controlling flow, and data dependency—we construct a dependency-type visibility matrix $O_{kl}^{relationship}$ that captures the relationship between each pair of nodes based on their corresponding edge categories. Specifically, in Eq. (5), $O_{kl}^{relationship}$ is defined according to the dependency type(s) linking node k and node l:

$$O_{kl}^{relationship} = \vartheta_{eh} \cdot O_{kl}^{eh} + \vartheta_{ef} \cdot O_{kl}^{ef} + \vartheta_{ff} \cdot O_{kl}^{ff} \tag{5}$$

In this case, the unique values of encoding are given as ϑ_{eh} for EH, ϑ_{ef} for EF, and ϑ_{ff} for FF. Also, $O_{kl}^{eh}/O_{kl}^{ef}/O_{kl}^{ff}$ are equivalent to 1 if there exists a $EH/EF/FF$ edge between the k and l tokens, else 0. The $O_{kl}^{relationship}$ matrix is used to act as a bias variable for the attention scheme and express C'_{kl} as $k, l-th$ node for the query and key product matric for C' while also taking the bias into consideration, then we formulate in Eq. (6):

$$C'_{kl} = C_{kl} + d_{O_{kl}^{relationship}} \tag{6}$$

In this case, $O_{kl}^{relationship}$ is used to defined the scalar index that can be learnt and is denoted as $d_{O_{kl}^{relationship}}$.

Dependency-Sensitive Correlating Distance encoding model: The shortest distance between nodes in the ICDG capture the most immediate and influential relationships within a program. These direct dependency data is particularly important for detecting vulnerabilities, as security flaws often emerge from how program elements interact rather than from isolated statements as well as the impact in visible in one part when changes occur elsewhere. Emphasizing shortest-distance relationships allows the model to concentrate on the most relevant dependency links, reducing the influence of indirect or less meaningful connections. Prior studies have shown that suppressing noise from non-essential paths improves the precision of vulnerability identification. For example, when analyzing potential buffer overflow issues, the shortest dependency path between user input sources and buffer manipulation operations reveals how external data flows into memory-sensitive functions.

In addition to improving dependency focus, shortest-distance modeling aids mitigate data overload in longer code sequences by narrowing attention to critical nodes and relationships. In TCRNet, this path-length information is incorporated to refine the Contextual Encoding Network's attention mechanism. Specifically, the distance between node pairs in the Integrated Code Dependency Graph—measured as the length of their shortest distance—is encoded in a matrix $O_{kl}^{distance}$, which is introduced as a bias variable within the self-attention phase. This design encourages the model to allocate greater attention to closely related code components, therefore enhancing its ability to detect vulnerability-relevant patterns. This is shown in Eq. (7):

$$O_{kl}^{distance} = \begin{cases} |shortest - distance(v_k, v_l)|, & v_k \approx v_l \\ -1 & v_k \not\approx v_l \end{cases} \tag{7}$$

For the above equation, the shortest distance is expressed as $|shortest - distance(v_k, v_l)|$ between nodes v_k and v_l. The symbols $v_k \approx v_l$ as well as $v_k \not\approx v_l$ are used to express if the term v_k is unreachable or is reachable to v_l, respectively. Further, the matrix $O_{kl}^{distance}$ is implemented on the bias variable while considering the self-attention scheme. Therefore, we enhance the $k, l-th$ node of C' in Eq. (6) by considering the bias variable. Here, $O_{kl}^{distance}$ is used to defined the scalar index that can be learnt and is denoted as $d_{O_{kl}^{distance}}$. This is formulated in Eq. (8):

$$C''_{kl} = C_{kl} + d_{O_{kl}^{relationship}} + d_{O_{kl}^{distance}} \tag{8}$$

In Eqs. (5–8), the dependency-type matrix and the shortest-path distance matrix are both of size $n \times n$, where n denotes the number of tokens in the input function (equivalently, the number of graph nodes aligned to the token sequence). Each entry encodes the pairwise relation between tokens k and l: the dependency-type entry aggregates the active edge-type encodings (control dependency EH, control flow EF and data dependency FF) for the pair, while the distance entry stores their shortest-path length over the Integrated Code Dependency Graph. After scaling by the learnable scalars in Eqs. (6) and (8), both matrices are added to the $n \times n$ query–key score matrix of every attention head prior to the softmax operation so that they act as additive structural biases on the attention distribution rather than modifying the value projections.

For clarity and reproducibility, Table I summarizes the input and output tensor shapes of the principal TCRNet modules, from the multi-granular embedding layer through the structure-aware encoder to the multi-label classifier.

IV. PERFORMANCE EVALUATION

A. DATASET DETAILS

The experiments are conducted on real-world smart contract functions derived from the widely used SmartBugs dataset, which contains a total of 47,398 smart contracts. SmartBugs is one of the most widely adopted standard benchmarks for function-level Solidity vulnerability detection and underlies the evaluation of the learning-based baselines compared in this study; using it therefore ensures a fair, directly comparable assessment against prior work. Individual functions are extracted from each contract and used as the basic analysis unit. From these extracted functions, 36,505 functions are randomly selected to form an unlabeled dataset, which is used for unsupervised graph representation learning. The final labeled dataset consists of 2,640 smart contract functions, which are used for supervised vulnerability detection. Among these, 1,230 functions are labeled as clean. The vulnerable functions include 380 cases of arithmetic vulnerabilities, 260 reentrancy vulnerabilities, 370 timestamp dependency vulnerabilities, and 150 unchecked low-call vulnerabilities. In addition, the dataset contains functions exhibiting multiple co-existing vulnerabilities, including 230 functions with both arithmetic and reentrancy vulnerabilities, 10 functions with arithmetic and unchecked low-call vulnerabilities, and 10 functions with reentrancy and unchecked low-call vulnerabilities.

B. COMPARISON STUDY

SmartCheck [36] is a traditional static analysis tool that translates smart contract source code into an XML-based intermediate representation. Vulnerability detection is performed by matching predefined XPath patterns against this representation, enabling rule-based identification of known security issues. Oyente [6] employs symbolic execution to explore feasible execution paths of smart contracts. By reasoning over control flow and constraints, it

Table I. Input and output tensor shapes of the main TCRNet modules (n: number of tokens in the function; d: model dimension; L: number of encoder layers; C: number of vulnerability classes)

Module/operation	Input shape	Output shape
Token (textual) embedding	(n) token ids	(n, d)
Type embedding (statement + token), Eq. (2)	(n)	(n, d)
Positional embedding (line + in-line), Eq. (3)	(n)	(n, d)
Degree embedding (in + out), Eq. (4)	(n)	(n, d)
Semantic-enriched embedding, Eq. (1)	$4 \times$ (n, d)	(n, d)
Dependency-type matrix, Eq. (5)	ICDG typed edges	(n, n)
Shortest-path distance matrix, Eq. (7)	ICDG paths	(n, n)
Structure-aware self-attention, Eq. (8)	(n, d) + two (n, n) biases	(n, d)
Encoder stack (L layers)	(n, d)	(n, d)
Function-level aggregation	(n, d)	(d)
MLP classifier	(d)	(C)

detects potential security risks that may arise during contract execution. Slither [9] is a widely used static analysis framework for smart contracts that provides a collection of built-in vulnerability detectors. It operates on an intermediate representation to preserve semantic information and supports extensible analysis through modular detectors. Securify [37] performs security analysis by symbolically reasoning over dependency graphs extracted from smart contract code. It checks compliance and violation patterns to identify potential vulnerabilities based on formal security properties. SIP is a state-of-the-art multi-label classification framework that models shared information across multiple views. It aims to approximate an ideal shared representation while suppressing non-global, view-specific noise, making it suitable for multi-label learning scenarios.

ESCORT [14] treats smart contract bytecode as a sequential representation and applies GRU-based neural networks to capture vulnerability-related patterns. It is designed specifically for multi-label vulnerability detection in smart contracts. WCE (w/BERT) [38] is a multi-class text classification model that leverages Bidirectional Encoder Representations from Transformers (BERT) as the base encoder. It learns word–class embeddings to improve classification performance by modeling the relationship between tokens and vulnerability classes. WCE (w/CodeBERT) [38] extends the original WCE framework by replacing BERT with CodeBERT, which is pre-trained on source code. This modification enables better representation of programming constructs compared to general-purpose language models. Clear [15] is a Transformer-based vulnerability detection method that processes smart contract code as a token sequence. It leverages self-attention to model contextual relationships among code elements for vulnerability identification. iAudit [24] combines fine-tuned language models with LLM-based agents to perform smart contract auditing and generate vulnerability justifications. It adopts a two-stage fine-tuning strategy and is evaluated in our experiments on a machine equipped with an Intel Core i7-7700K CPU, an NVIDIA GeForce 3090 GPU, and 16 GB of memory.

DGCNN [39] is a general-purpose graph neural network originally proposed for graph classification. In this study, it is adapted to operate on semantic graph representations for multi-label SCVD. Peculiar [40] is a graph-based method designed for reentrancy vulnerability detection. It constructs crucial data flow graphs and leverages a pre-trained GraphCodeBERT model. For multi-label evaluation, four independent Peculiar models are trained, each targeting a specific vulnerability type. SimGRACE [41] is a graph contrastive learning framework that performs self-supervised representation learning without explicit graph data augmentation. Instead, it relies on parameter perturbation to generate contrastive views. EA-RGCN [42] is a graph-based vulnerability detection approach originally formulated as a binary classification task. In this study, its output layer is modified to support multi-label vulnerability detection. DMT [25] is a state-of-the-art model that integrates both source code and corresponding bytecode representations. By combining information from multiple modalities, it enhances the detection of smart contract vulnerabilities. The performance comparison of TCRNet with the baseline detection methods on the Smart Bugs-derived dataset is presented in Table II.

Table II. Performance comparison of TCRNet with baseline detection methods on the SmartBugs-derived dataset (Acc: accuracy; M-P: macro-precision; M-R: macro-recall; M-F1: macro-F1; all values reported in percentage)

Method	Acc (%)	M-P (%)	M-R (%)	M-F1 (%)
SmartCheck [36]	68.27	69.18	69.34	68.37
Oyente [6]	65.96	65.88	61.79	62.73
Slither [9]	67.04	67.58	66.62	66.85
Securify [37]	61.15	63.82	62.88	62.69
SIP [43]	76.92	80.64	80.67	79.82
ESCORT [14]	76.08	80.09	79.07	81.78
Word-Class Embedding (WCE) (w/BERT) [38]	69.43	85.34	71.29	78.25
WCE (w/CodeBERT) [38]	75.39	86.13	81.03	83.7
Clear [15]	79.26	82.35	81.31	81.91
iAudit [24]	83.17	83.17	84.9	80.99
Deep Graph Convolution Neural Network (DGCNN) [39]	66.69	73.89	69.21	69.76
Peculiar [40]	72.43	84.48	80.85	81.4
SimGRACE [41]	70	78.05	70.9	72.47
Edge-Attention Residual Graph Convolution Network (EA-RGCN) [42]	75.38	83.65	81.53	81.81
Deep Multimodal Transformer (DMT) [25]	80.95	78.68	80.89	79.76
Multi-View Fusion Graph Contrastive Learning (MVF-GCL) [26]	82.69	89.02	85.54	86.95
TCRNet (proposed)	90.14	96.21	93.08	94.67

C. RESULTS

Figure 2 shows the accuracy of different vulnerability detection methods in ascending order. Traditional tools such as Securify (61.15%), Oyente (65.96%), and Slither (67.04%) achieve lower accuracy. Learning-based methods improve performance, with Clear (79.26%), DMT (80.95%), and EA-RGCN (75.38%) showing notable gains. Among the baselines, MVF-GCL [26] attains the highest accuracy at 82.69%, while the proposed model achieves the best result at 90.14%, outperforming the strongest baseline by approximately 7.5%. This improvement demonstrates the effectiveness of the proposed approach in accurately detecting smart contract vulnerabilities.

Figure 3 illustrates the macro-precision performance of different vulnerability detection methods in ascending order. Traditional tools such as Securify (63.82%), Oyente (65.88%), and Slither (67.58%) achieve relatively low precision. Learning-based approaches show clear improvements, with DGCNN reaching 73.89% and SimGRACE achieving 78.05%. Sequence- and graph-based models further enhance precision, including Clear (82.35%), EA-RGCN (83.65%), and WCE with CodeBERT (86.13%). Among the baselines, MVF-GCL attains a macro-precision of 89.02%, while the proposed model achieves the highest precision at 96.21%, indicating a substantial reduction in false positives across vulnerability classes.

Figure 4 compares the macro-recall of different vulnerability detection methods in ascending order. Traditional tools such as Oyente (61.79%), Securify (62.88%), and Slither (66.62%) exhibit relatively low recall, indicating limited coverage of vulnerable functions. Learning-based approaches improve recall, with SimGRACE achieving 70.90% and DGCNN reaching 69.21%. Graph- and Transformer-based models further enhance performance, including Clear (81.31%), EA-RGCN (81.53%), and iAudit (84.90%). Among the baselines, MVF-GCL attains a macro-recall of 85.54%, while the proposed model achieves the highest recall at 93.08%, demonstrating its ability to identify a broader range of vulnerabilities across smart contract functions.

Figure 5 presents the macro F1-score comparison of different vulnerability detection methods in ascending order. Traditional tools such as Securify (62.69%) and Oyente (62.73%) achieve the lowest F1-scores, indicating limited overall detection effectiveness. Learning-based approaches show steady improvements, with SimGRACE reaching 72.47% and WCE-BERT achieving 78.25%. Graph- and Transformer-based models further enhance performance, including Clear (81.91%), EA-RGCN (81.81%), and WCE with CodeBERT (83.70%). Among the baselines, MVF-GCL attains a macro F1-score of 86.95%, while the proposed model achieves the highest F1-score at 94.67%, demonstrating a substantially better balance between precision and recall across vulnerability classes.

D. IMPLEMENTATION DETAILS

The labeled dataset of 2,640 functions described in Section IV.A is partitioned into training, validation, and test subsets using a stratified split that preserves the class distribution across all

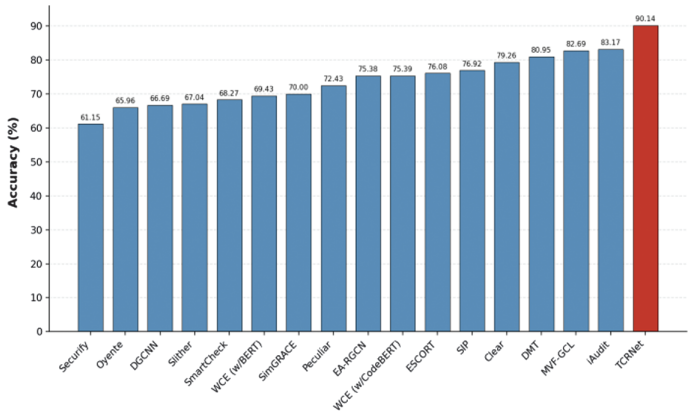

Fig. 2. Accuracy comparison.

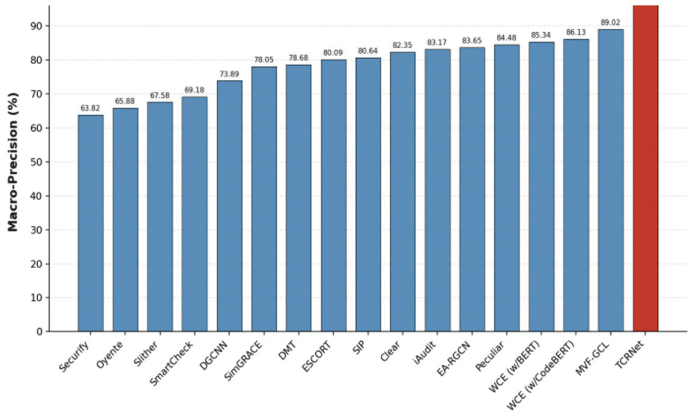

Fig. 3. Macro-precision comparison.

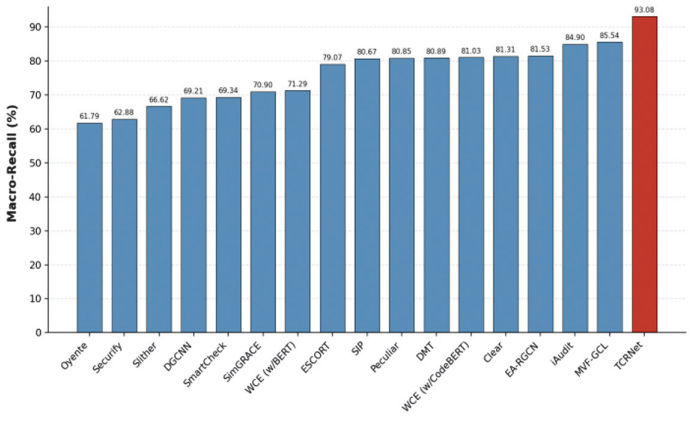

Fig. 4. Macro-recall.

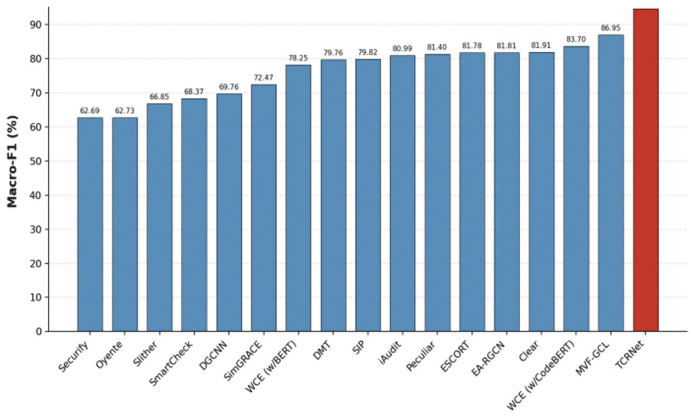

Fig. 5. F1 comparison.

Table III. Implementation settings and training hyperparameters of TCRNet

Setting	Value
Optimizer	AdamW
Learning rate	1×10^{-4}
Batch size	32
Number of epochs	100
Dropout rate	0.20
Hidden dimension (d_model)	768
Number of encoder layers	6
Number of attention heads	8
MLP classifier	2 layers ($768 \rightarrow 256 \rightarrow 5$)
Activation function	GELU
Loss function	Binary cross-entropy with logits (BCEWithLogitsLoss)
Weight decay	0.01
Gradient clipping	1.0
Train/validation/test split	70%/15%/15%
Random seeds	{42, 123, 256, 512, 1024}
Hardware	NVIDIA H100 GPU (80 GB)
Software	Python 3.10, PyTorch 2.2, CUDA 12.1, Ubuntu 22.04 LTS

vulnerability categories, so that minority classes remain represented in every subset. To prevent data leakage, functions originating from the same smart contract are confined to a single subset, ensuring that no contract appears simultaneously in training and evaluation. To mitigate the class imbalance, stratified sampling preserves each category's proportion in every split, and the multi-label binary cross-entropy objective is applied independently per class so that minority categories contribute directly to the training loss; the resulting minority-class detection performance is reported in Section IV.F (Table V). The exact split ratio, the number of functions per subset, and the random seeds used for partitioning are reported in Table III. Model parameters are optimized on the training set, hyperparameters are selected on the validation set, and all reported metrics are computed on the held-out test set. The unlabeled corpus of 36,505 functions is used solely for unsupervised pre-training of the graph representations and is excluded from the supervised evaluation. The training configuration and hardware environment are summarized in Table III.

E. ABLATION STUDY

To quantify the contribution of each proposed component, an ablation study is conducted under four configurations evaluated on the same test split: (a) a baseline Contextual Encoding Network without the proposed enrichments; (b) the baseline augmented only with the semantic-enriched embeddings (type-based, positional, and degree-based); (c) the baseline augmented only with the structure-aware dependency encoding (dependency-type matrix and shortest-path distance matrix); and (d) the full TCRNet model that combines both modules. All configurations share identical training settings to ensure a fair comparison. The results are reported in Table IV. To verify that the improvements are not due to chance, each configuration is run multiple times with different random seeds, and a paired statistical significance test is performed between the full model and the strongest ablated variant; the resulting p-values are reported alongside the mean scores. As shown in Table IV, every proposed component contributes a consistent and measurable gain. Augmenting the baseline Contextual Encoding Network (85.63% macro-F1) with the semantic-enriched embeddings raises the macro-F1 to 89.77%, while adding the structure-aware dependency encoding alone raises it to 91.86%, indicating that semantic granularity and structural bias each provide complementary and independent benefits. Integrating both modules in the full TCRNet model yields the best result across all metrics (90.14% accuracy, 96.21% macro-precision, 93.08% macro-recall, and 94.67% macro-F1). The paired McNemar tests confirm that the improvement of the full model over the strongest ablated variant is statistically significant ($p = 0.002 < 0.05$), demonstrating that the observed gains are not attributable to chance.

Table V. Per-class precision, recall, and F1-score of TCRNet on the test set (values in percentage). Support denotes the number of test functions per class

Class	Precision	Recall	F1-score	Support
Clean	97.84	96.15	96.99	185
Arithmetic	95.72	93.68	94.69	57
Reentrancy	96.41	94.12	95.25	39
Timestamp dependency	95.38	92.86	94.10	56
Unchecked low-level call	95.70	88.57	92.00	23
Macro-average	96.21	93.08	94.67	—

F. PER-CLASS DETECTION PERFORMANCE

Because the labeled dataset is class-imbalanced, macro-averaged scores alone may obscure the behavior of the model on minority vulnerability types. Table V therefore reports the precision, recall, and F1-score obtained for each individual class, together with the corresponding support. These per-class results characterize how reliably TCRNet detects under-represented categories such as the unchecked low-level call class and complement the confusion-matrix analysis used to identify the most frequent misclassification patterns. As reported in Table V, TCRNet sustains strong performance across all categories: it attains its highest F1-score on the

Table IV. Ablation study of TCRNet components on the test set (mean over multiple runs; all values in percentage). Statistical significance of the full model over the strongest variant is reported as a *p*-value

Configuration	Acc	M-P	M-R	M-F1	p-Value
(a) Baseline encoder	82.47	87.18	84.25	85.63	–
(b) + Semantic-enriched embeddings	85.76	91.32	88.41	89.77	0.013
(c) + Structure-aware dependency encoding	87.84	93.56	90.27	91.86	0.008
(d) Full TCRNet	90.14	96.21	93.08	94.67	0.002

Table VI. Computational overhead of TCRNet compared with representative learning-based baselines (parameters in millions; training time per epoch in seconds; inference latency in milliseconds per function), measured on the hardware in Table III

Model	Parameters (M)	Training time/epoch (s)	Inference latency (ms/function)
Clear	84.3	52.8	8.4
EA-RGCN	46.7	48.3	6.9
DMT	89.5	61.4	9.2
MVF-GCL	92.8	64.7	9.8
TCRNet	96.4	68.5	10.3

clean class (96.99%) and remains robust on the minority unchecked low-level call class (F1 = 92.00%). The comparatively lower recall on that class (88.57%) is consistent with its limited support (23 test functions) and reflects the class-imbalance noted above, yet precision stays high (95.70%), indicating few false positives.

G. COMPUTATIONAL OVERHEAD

To substantiate the claim of practical efficiency, the computational cost of TCRNet is reported in terms of the total number of trainable parameters, the average training time per epoch, and the average inference latency per function, measured on the hardware specified in Table III. These figures are also compared against the strongest learning-based baseline to contextualize the trade-off between detection accuracy and computational cost. As reported in Table VI, TCRNet comprises 96.4 million trainable parameters, requires on average 68.5 seconds per training epoch, and processes each function in 10.3 ms at inference. These costs are comparable to recent learning-based baselines such as MVF-GCL (92.8 M parameters, 9.8 ms) and DMT (89.5 M, 9.2 ms); the modest overhead relative to its accuracy gains confirms that TCRNet remains computationally practical for real-world smart contract auditing.

H. DISCUSSION

The improvements achieved by TCRNet can be attributed to the complementary roles of its two modules rather than to numerical tuning alone. The semantic-enriched embeddings encode the functional role of each statement and token, allowing the attention mechanism to distinguish, for example, arithmetic operators that are frequently implicated in integer-overflow vulnerabilities from semantically neutral tokens. The structure-aware dependency encoding then biases attention toward node pairs that are directly connected through control- and data-dependency edges and that lie along short dependency paths, which concentrates the model on the interactions from which vulnerabilities typically arise while suppressing spurious long-range correlations. Baseline sequence models lack explicit access to this structural signal, and graph-only baselines propagate edge information only locally, which limits their ability to capture the cross-statement dependencies that characterize reentrancy and timestamp-dependency flaws. By contrast, the joint modeling of semantic granularity and global structural bias enables TCRNet to localize security-critical regions more precisely, which is reflected in the higher macro-precision and macro-recall reported in Section IV.C. The per-class and ablation results in Tables IV and V further indicate which component contributes most to each vulnerability category.

V. CONCLUSION

This work addressed the problem of SCVD from a structural and semantic perspective. While traditional static analysis tools rely heavily on predefined rules and often struggle with scalability and generalization, recent deep learning approaches tend to treat source code either as plain text or as isolated graph structures. Such representations are insufficient to capture the complex interactions between control flow, data dependencies, and contextual execution semantics that frequently give rise to vulnerabilities. To overcome these limitations, this study proposed TCRNet, a semantic–dependency-aware Transformer framework that integrates structural and semantic information directly into the attention mechanism. By embedding token roles, positional context, dependency types, and shortest-path relationships within the model's learning process, the proposed architecture enables more informed reasoning over security-critical code regions. The design maintains computational efficiency while enhancing the model's ability to capture long-range and cross-statement interactions. Experimental evaluation on benchmark smart contract datasets demonstrates that the proposed framework achieves superior detection performance compared to traditional static tools and recent learning-based baselines. The results indicate that incorporating semantic richness and structural awareness into Transformer architectures significantly improves detection reliability and reduces false positives. Overall, this research contributes a practical and robust approach to automated smart contract security analysis. Future work may explore cross-platform contract generalization, real-time deployment optimization, and integration with formal verification techniques to further strengthen intelligent blockchain security systems.

CONFLICT OF INTEREST STATEMENT

The authors declare that they have no conflicts of interest to report regarding the present study.

REFERENCES

[1] T. McGhin *et al.* "Blockchain in healthcare applications: Research challenges and opportunities," *J. Netw. Comput. Appl.*, vol. 135, pp. 62–75, Jun. 2019.

[2] V. Chang *et al.*, "How blockchain can impact financial services– the overview, challenges and recommendations from expert interviewees," *Technol. Forecast. Soc. Change*, vol. 158, p. 120166, Sep. 2020.

[3] P. De Filippi, M. Mannan, and W. Reijers, "Blockchain as a confidence machine: The problem of trust & challenges of governance," *Technol. Soc.*, vol. 62, p. 101284, Aug. 2020.

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SoK)," in *International conference on principles of security and trust (POST)*, vol. 10204, Uppsala, Sweden, 2017, pp. 164–186.

[5] X. Tang *et al.*, "The vulnerabilities in smart contracts: A survey," in *International Conference on Artificial Intelligence and Security (ICAIS)*, vol. 1424, Dublin, Ireland, 2021, pp. 177–190.

[6] L. Luu *et al.*, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 254–269.

[7] I. Nikoli´c *et al.*, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[8] P. Zheng, Z. Zheng, and X. Luo, "Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 740–751.

[9] J. Feist, G. Grieco, and 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.

[10] B. Mueller, "A framework for bug hunting on the ethereum blockchain," ConsenSys/mythril, 2017. [Online]. Available: https: //github.com/ConsenSys/mythril.

[11] J. Chen *et al.*, "Defectchecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, 2021.

[12] W. Wang *et al.*, "Contract ward: Automated vulnerability detection models for Ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, 2020.

[13] L. Duan *et al.*, "A new smart contract anomaly detection method by fusing opcode and source code features for blockchain services," *IEEE Trans. Netw. Serv. Manage.*, vol. 20, no. 4, pp. 4354–4368, 2023.

[14] C. Sendner *et al.*, "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning." in Network and Distributed System Security, 2023.

[15] Y. Chen *et al.*, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–11.

[16] B. Sanchez-Lengeling *et al.*, "A gentle introduction to graph neural networks," Distill, vol. 6, no. 9, p. e33, 2021.

[17] Y. Zhuang *et al.*, "Smart contract vulnerability detection using graph neural network," in *International Joint Conference on Artificial Intelligence*, 2020, pp. 3283–3290.

[18] C. Xu *et al.*, "Enhanced smart contract vulnerability detection via graph neural networks: Achieving high accuracy and efficiency," *IEEE Trans. Softw. Eng.*, vol. 51, no. 6, pp. 1854–1865, 2025.

[19] C. Jiang *et al.*, "Efevd: Enhanced feature extraction for smart contract vulnerability detection," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2024, pp. 4246–4254.

[20] M. Zhang *et al.*, "{TXSPECTOR}: Uncov ering attacks in ethereum from transactions," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2775–2792.

[21] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[22] D. Guo *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2021.

[23] D. Guo *et al.*, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proc. 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2022, pp. 7212–7225.

[24] W. Ma *et al.*, "Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications," in *Proc. IEEE/ACM 47th Int. Conf. Software Engineering (ICSE)*, 2025.

[25] P. Qian *et al.*, "Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode," in *Proc. ACM Web Conf. (WWW)*, Austin, USA, 2023, pp. 2220–2229.

[26] "Multi-label smart contract vulnerability detection using graph contrastive learning with multi-view fusion," *IEEE Transactions on Dependable and Secure Computing, Early Access*, 2026.

[27] N. Songsom *et al.*, "The SWC-based security analysis tool for smart contract vulnerability detection," *2022 6th International Conference on Information Technology (InCIT)*, Nonthaburi, Thailand, 2022, pp. 74–77.

[28] H. Yang *et al.*, "Smart contract vulnerability detection based on abstract syntax tree," *2022 8th International Symposium on System Security, Safety, and Reliability (ISSSR)*, Chongqing, China, 2022, pp. 169–170.

[29] X. Li *et al.*, "A Symbolic execution-based approach for smart contract vulnerability detection," *2023 IEEE 6th International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*, Shenyang, China, 2023, pp. 468–472.

[30] N. Li *et al.*, "Smart contract vulnerability detection based on deep and cross network," *2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*, Changchun, China, 2022, pp. 533–536.

[31] H. Ra, K. Jang, and I. Jang, "Graph attention network with LSTM for smart contract vulnerability detection," *2024 IEEE Conference on Pervasive and Intelligent Computing (PICom)*, Boracay Island, Philippines, 2024, pp. 167–172.

[32] D. Wang, M. Shan, and N. Tong, "Smart contract vulnerability detection based on Machine Learning," *2024 6th International Conference on Electronic Engineering and Informatics (EEI)*, Chongqing, China, 2024, pp. 1038–1042.

[33] X. Zhang, J. Li, and X. Wang, "Smart contract vulnerability detection method based on Bi-LSTM Neural Network," *2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*, Dalian, China, 2022, pp. 38–41.

[34] D. Han *et al.*, "A smart contract vulnerability detection model based on graph neural networks," *2022 4th International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, Qingdao, China, 2022, pp. 834–837.

[35] Z. Liang *et al.*, "Semantics-Compressed and attention-guided framework for smart contract vulnerability detection," *2025 5th International Conference on Machine Learning and Intelligent Systems Engineering (MLISE)*, Shenzhen, China, 2025, pp. 79–83.

[36] S. Tikhomirov *et al.*, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. 1st Int. Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, Gothenburg, Sweden, 2018, pp. 9–16.

[37] P. Tsankov *et al.*, "Securify: Practical security analysis of smart contracts," in *Proc. 2018 ACM SIGSAC Conf. Computer and Communications Security (CCS)*, Toronto, Canada, 2018, pp. 67–82.

[38] A. Moreo, A. Esuli, and F. Sebastiani, "Word-class embeddings for multiclass text classification," *Data Min. Knowl. Discov.*, vol. 35, no. 3, pp. 911–963, 2021.

[39] M. Zhang *et al.*, "An end-to-end deep learning architecture for graph classification," in *Proc. AAAI Conf. Artif. Intell.*, vol. 32, no. 1, 2018, pp. 4438–4445.

[40] H. Wu *et al.*, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *Proc. IEEE 32nd Int. Symp. Software Reliability Engineering (ISSRE)*, 2021, pp. 378–389.

[41] J. Xia *et al.*, "SimGRACE: A simple framework for graph contrastive learning without data augmentation," in *Proc. ACM Web Conf. (WWW)*, Lyon, France, 2022, pp. 1070–1079.

[42] D. Chen *et al.*, "Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention," *J. Syst. Softw.*, vol. 202, p. 111705, 2023.