MFVD: A Multimodal Feature-Based Vulnerability Detection Framework for Smart Contracts

Tingxun Chen[1], Bo Yin[1,*], Yuan Zhu[1], Yuchuan Chen[1]
[1]School of Computer Science, Changsha University of Science and Technology, Changsha, China
xun@stu.csust.edu.cn, yinbo@hnu.edu.cn, zhuy@stu.csust.edu.cn, chuan990621@163.com

***Abstract*—Smart contract vulnerability detection (SCVD) has attracted increasing attention in both academia and industry. Pre-trained language models (PLMs) are trained on large-scale data, have shown strong generalization ability. However, existing PLMs-based SCVD methods are typically limited to processing inputs of 512 or 1024 tokens, or lack structural information, which degrades vulnerability detection performance. An important question arises: How effective are existing PLMs when applied to SCVD? In this paper, we introduce a new multimodal feature-based vulnerability detection framework named MFVD. We utilize segmentation encoding strategy and graph presentation to tackle the limitations above. MFVD enhances UniXcoder to process multimodal data—the source code sequence and the data flow graph—and enables binary classification. Extensive experiments demonstrate that MFVD outperforms state-of-the-art methods in recall, precision, and F1 scores, as well as in the stability of these metrics, achieving 96.14% F1 for reentrancy vulnerabilities and 96.71% F1 for timestamp dependency vulnerabilities. We also evaluate 11 PLMs on source code, with results indicating that UniXcoder outperforms the others in most scenarios.**

***Index Terms*—pre-trained language models, vulnerability detection, smart contracts, multimodal learning.**

I. Introduction

Smart contracts are executable programs deployed on blockchains and often manage assets of significant economic value. Due to the immutability of blockchains, vulnerabilities in deployed contracts cannot be modified, potentially leading to severe financial losses and systemic risks. For example, in 2016, the well-known DAO attack resulted in the theft of over $50 million of cryptocurrency and led to a hard fork of the Ethereum network [1]. Therefore, *smart contract vulnerability detection* (SCVD) has become the research focus of academia and industry.

Existing methods for SCVD can be categorized into two main types, traditional program analysis–based methods and machine learning & deep learning–based methods. The former employs traditional program analysis techniques that identify potential vulnerabilities by simulating contract execution paths [1], using manually predefined rules [2, 3], or verifying whether specific security specifications are met [4]. However, such methods often suffer from path explosion and low detection efficiency, especially for large and complex smart contracts. While ContractWard [5] utilizes machine learning techniques, such as XGBoost, random forest and so on. Recent studies adopt deep learning techniques, including graph neural networks such as DR-GCN and TMP [6], semantic–syntactic similarity modeling [7] and contrastive learning [8] to improve vulnerability detection. Peculiar [9] firstly introduced PLM GraphCodeBERT for reentrancy vulnerability and is based on crucial data flow graphs. ReVulDL [10] further extended Peculiar to vulnerability localization. DMT [11] proposes a cross-modal framework including PLM called BERT to learn representations source code and bytecode.

Smart contracts typically contain a large amount of code and complex logic, which increases the difficulty of automated vulnerability analysis. In Table I, we summarize statistics from the SmartBugs Wild Dataset (SWD) [12], showing that nearly 10.1% of smart contracts contain 200+ lines of code, while about 43.3% exceed 512 tokens after tokenization. Pre-trained language models (PLMs) follow a pre-training and fine-tuning paradigm and have demonstrated strong generalization ability in code understanding. However, existing PLM-based methods still face two major limitations. First, PLMs are limited by fixed input lengths (512 or 1024 tokens), which leads to code truncation in long code sequences; Methods such as Peculiar [9] and DMT [11] do not recognize this issue, leading to the truncation of long code sequences and the loss of complete semantic information. Second, some methods like LineVul [13] treat source code only as token sequences and ignore code structural information, limiting their ability to capture deeper program semantics. More comprehensive code representations in PLMs are more effective for code understanding.

TABLE I: Distribution of smart contracts in [12]

Code lines			**Tokens**		
Range (#lines)	#Contracts	#Contracts (%)	Range (#tokens)	#Contracts	#Contracts (%)
1–200	182,902	89.9	1–512	115,429	56.7
200–500	16,624	8.1	512–1024	47,023	23.1
500+	4,187	2.0	1024+	41,261	20.2
Total	203,713	100.0	Total	203,713	100.0

In this paper, we propose MFVD, a multimodal framework for SCVD that integrates token sequences and graph-based representations using the PLM UniXcoder [14]. To address code truncation, we adopt a segmentation encoding strategy that splits long token sequences into manageable segments, whose representations are aggregated into statement-level embeddings and further fused to preserve whole-program

* Corresponding author.
[1]https://en.wikipedia.org/wiki/The_DAO

semantics. To address the loss of structural information, vulnerability-specific data flow graphs are processed with graph-guided masks and multi-head attention, enabling PLMs to learn code data-flow structures effectively. We evaluated MFVD on two vulnerabilities: *reentrancy vulnerability* and *timestamp dependency vulnerability* and compared with 12 existing methods. The experimental results indicate that MFVD outperforms state-of-the-art methods in terms of recall, precision, and F1 scores. Specifically, the F1 scores achieved are 96.14% for detecting reentrancy vulnerabilities and 96.71% for detecting timestamp dependency vulnerabilities.

Moreover, with the advancement of PLMs, an important question arises: How effective are the existing PLMs when applied to SCVD? Various PLMs have been proposed for source code understanding. BERT [15] and RoBERTa [16] treat code as token sequences, while CodeBERT [17] uses joint pre-training on code and natural language. ELECTRA [18] adopts a discriminator–generator paradigm, and GraphCodeBERT [19] incorporates data flow graph to enhance structuralmodeling. Recent PLMs including CodeT5 [20], CodeT5+ [21], LongCoder [22], PLBART [23], and UniXcoder [14], diverse code-related tasks and long code sequences by modeling source code, comments, or syntax structures. No previous research has experimentally compared various PLMs on SCVD. In this paper, we conduct an empirical evaluation by comparing with other 10 PLMs in segmentation encoding strategy.

The main contributions are summarized as follows.

- We propose MFVD, a UniXcoder-based framework fine-tuned for SCVD, and conduct it through extensive experiments on reentrancy and timestamp dependency vulnerabilities.
- We solve the code truncation loss of structural information issues inherent for PLMs when processing the source code. MFVD combines sequence representations and graph representations of source code to enhance code representation.
- To the best of our knowledge, this study is the first to empirically compare existing PLMs regarding the SCVD problem.

II. MFVD METHOD

A. Overview

Figure 1 illustrates the architecture of MFVD, which consists of two modules: a source code processing module and a graph processing module. To better learn code semantics, we use UniXcoder to extract feature representations from two modalities. We adopt two UniXcoder encoders with different roles: TE-UXEncoder encodes token-level representations, while ME-UXEncoder aggregates segment representations in the segmentation encoding strategy.

Algorithm 1 outlines the pseudo-code for the MFVD method. The source code processing module (lines 1–2) generates the feature representation $\hat{X}^C$ of sc using the CodeExtraction() function (detailed in Algorithm 2). The graph processing module (lines 3–6) processes cg_{seq} and sc_{seq} using positional encoding and graph-guided mask attention to capture data dependencies within its graph structure, resulting in the feature representation $\hat{X}^G$. The above description is presented in Section II-C2. The fused representation X (line 7) is projected by a linear layer and fed into a Softmax function to produce the final vulnerability label $label$ (line 8).

The vulnerability detection on MFVD comprises three stages: *data preparation, feature extraction, and vulnerability detection.* In the following, we provide a detailed description of each stage.

Algorithm 1: MFVD

Input: the source code sequence sc and the graph dfg of a smart contract, and the length limit len
Output: the label $label$ of this smart contract

1 $sc_{seq} \leftarrow$ **Tokenization**(sc);
2 $\hat{X}^C \leftarrow$**CodeExtraction**(sc_{seq}, len);
3 $cg_{seq} \leftarrow$ **Tokenization**(dfg);
4 $cg_{seq} \leftarrow$**PositionEncoding**(sc_{seq}, cg_{seq});
5 $cg_{seq} \leftarrow$**MaskedAttention**(cg_{seq});
6 $\hat{X}^G \leftarrow$**TE-UXEncoder**(cg_{seq});
7 $X \leftarrow$**Concat**($\hat{X}^C$, $\hat{X}^G$);
8 $label \leftarrow$**Softmax**(**Linear**(X));

B. Data preparation

We utilize the smart contract dataset from SWD [12] and ESC [6], as described in Section III-A. The source code is preprocessed by removing documentation comments and normalizing blank lines to reduce noise during tokenization and feature extraction.

We use $SC = \{s_1, s_2, ..., s_n\}$ to denote source code, where $s_i \in SC$ represents an element in the code, such as a symbol or statement. Since the vulnerabilities we study are closely related to the state variables in the code, to capture the data and variable dependencies within SC, we adopt a data flow graph [19]. Specifically, we parse smart contract code into an AST using Tree-Sitter with Solidity grammar rules [2]. Variables are extracted from identifier nodes in the AST and denoted as $R = \{r_1, r_2, \ldots, r_k\}$, typically represented as identifiers in the leaf nodes. Based on the AST, we construct a data flow graph $G(SC) = (V, E)$ in which variable $r_i \in R$ is represented by node v, and edges $e(v_i, v_j)$ encode dependency relationships between variables derived from the AST, which include data-flow dependencies and computation dependencies.

However, data flow graph contains structural information irrelevant to vulnerability detection. To address this, we perform vulnerability-driven graph refinement by retaining only nodes and edges related to vulnerability-specific variable keywords. Specifically we focus on **call.value** which can lead to reentrancy vulnerability by triggering external contract callbacks before state updates, and **block.timestamp** which can lead to timestamp dependency vulnerability by making code susceptible to block time manipulation by attackers. We identify nodes corresponding to these variable keywords

[2]https://tree-sitter.github.io/tree-sitter

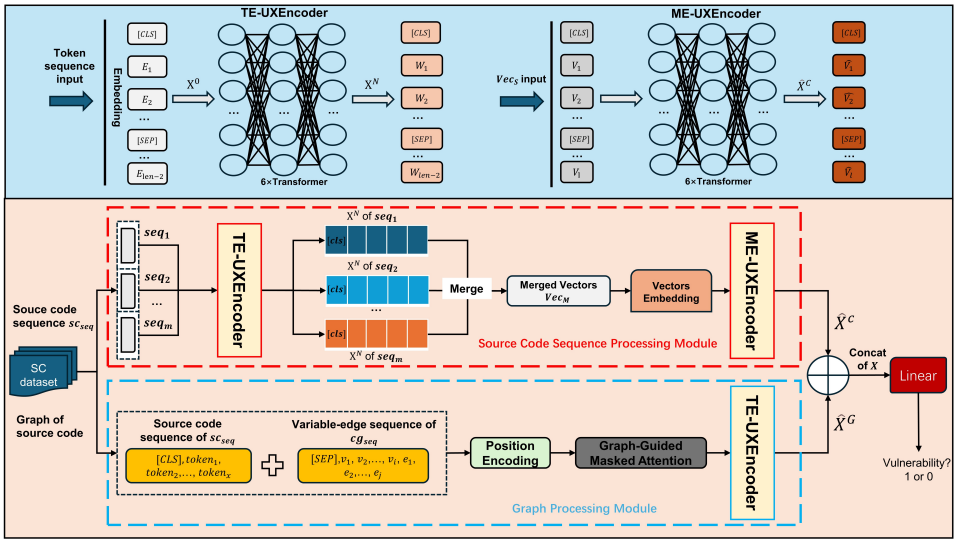

Fig. 1: MFVD architecture

and their directly or indirectly related edges. We denoted the refined graph as dfg.

Finally, the source code sequence sc and the graph dfg are transformed to sc_{seq} and cg_{seq} via tokenization, respectively. sc_{seq} is the input of the source code sequence processing module. Additionally, cg_{seq} will be merged with variable keywords of sc_{seq} and used as input for the graph processing module.

C. MFVD structure

1) UniXcoder-Encoder: As shown in Figure 1, UniXcoder-Encoder has two variants with identical structures: TE-UXEncoder and ME-UXEncoder, designed for different data types. In UniXcoder embedding layer, token sequence with n tokens will be converted into n numbers of 768-dimensional vectors, which be represented as X^0 in Figure 1. X^0 is then passed through N (set to 6) transformer encoder layers, generating a context-aware representation X^N. For the n-th layer of the transformer encoder with the input vector X^{n-1}, the multi-head self-attention operation produces a vector X^n, computed as $X^n = transformer_n(X^{n-1})$ for $n \in [1, N]$. The details are as follows:

$$H^n = \mathrm{LN}\big(\mathrm{MultiAttn}(X^{n-1}) + X^{n-1}\big) \quad (1)$$

$$X^n = \mathrm{LN}\big(\mathrm{FFN}(H^n) + H^n\big) \quad (2)$$

where MultiAttn() stands for multi-head self-attention mechanism, FFN() denotes a two-layer feed-forward neural network, and LN() is a normalization operation.

2) Feature extraction: This section presents a detailed description of the feature extraction process of the two modules. **Source code sequence feature extraction.** In Algorithm 2, we employ the segmentation encoding strategy (Algorithm 1, line 2) that divides the input token sequence sc_{seq} into multiple segments $\{seq_1, \ldots, seq_m\}$ according to len. Each seq_i is sequentially passed through TE-UXEncoder to generate the feature representation X^N (line 7). The special sequence start token cls is used to aggregate the semantic information of a segment. We merge all token sequences seq_i into $Alltoken$ (line 10) and apply average pooling over all cls tokens to obtain a global context vector CLS (line 9,11). By prepending CLS to $Alltoken$ and applying padding, we construct the merged representation Vec_M (line 12). Then VectorsEmbedding maps Vec_M to Vec_S, which is then fed into ME-UXEncoder to generate the final code representation $\hat{X}^C$ (lines 13-14).

Algorithm 2: CodeExtraction()

Input: Word segment sequence sc_{seq} and the length limit len
Output: Feature sequence $\hat{X}^C$

1 **if** $length(sc_{seq}) > len$ **then**
2 $m \leftarrow \lceil \frac{\text{length}(sc_{seq})}{len} \rceil$;
3 Split sc_{seq} into m segments $\{seq_1, \ldots, seq_m\}$;
4 $Alltoken \leftarrow []$;
5 $CLS \leftarrow []$;
6 **foreach** $seq_i \in \{seq_1, \ldots, seq_m\}$ **do**
7 $X^N \leftarrow$ **TE-UXEncoder**(seq_i);
8 $cls \leftarrow X^N[0]$;
9 Append cls to CLS;
10 Append X^N to $Alltoken$;
11 $CLS \leftarrow$ **MeanPooling**(CLS);
12 $Vec_M \leftarrow$ **Merge**$(Alltoken, CLS)$;
13 $Vec_S \leftarrow$ **VectorsEmbedding**(Vec_M);
14 $\hat{X}^C \leftarrow$ **ME-UXEncoder**(Vec_S);
15 **else**
16 $\hat{X}^C \leftarrow$ **TE-UXEncoder**(sc_{seq});

Limitations of code truncation. Although segmentation can handle long code sequences, it may disrupt the integrity of source code sequence. VectorsEmbedding (line 13) can address this by transforming discrete token representations into statement-level representations through the aggregation of token vectors after segmentation. As shown in Figure 2, we construct a relation matrix $L \in \mathbb{R}^{l \times t}$ that captures

the relationship between source code statements and tokens, where l is the number of statements and t is the number of tokens post-tokenization. If a token in column j belongs to a statement in row i, then $L_{ij} = 1$; otherwise, it is 0. The token vectors Vec_M obtained from segmentation encoding form a vector matrix $T \in \mathbb{R}^{t \times d_h}$, where d_h is the dimensionality of a single token vector. By performing the matrix multiplication $L \times T$, we derive the statement-level representations, which essentially average the token vectors contained in each statement. This results in fine-grained statement vector representations Vec_S.

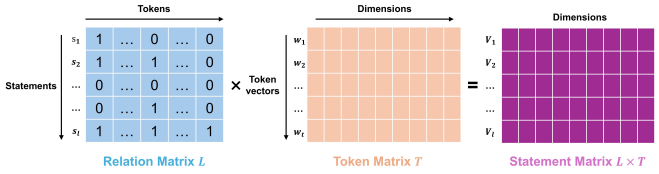

Fig. 2: VectorsEmbedding

Graph feature extraction. The obtained graph in Section II-B is represented as $I = \{[CLS], sc_{seq}, [SEP], V', E'\}$, where V' and E' represent cg_{seq}. $[CLS]$ denotes the start token, and $[SEP]$ denotes the separator token. To incorporate the graph structure into the transformer model, we utilize the approach described in [9]. (1) Position encoding allows the PLM to recognize the relative hierarchy or proximity of variables within the graph, determining whether a variable is close to a significant node. (2) graph-guided masked attention captures structural associations between nodes, focusing the model's attention on pairs of nodes in the graph that capture data dependencies, let $M \in \{0,1\}^{c \times c}$ be the mask matrix for the graph nodes, where c is the number of nodes. The element M_{ij} in the mask matrix is set to 1 when the node pair (v_i, v_j) is connected in the graph, and 0 otherwise. Finally, the graph processing module outputs the feature vector $\hat{X}^G$ of the graph.

D. Vulnerability detection

Finally, we utilize two data modalities for feature representation: source code sequence $\hat{X}^C$ and graph $\hat{X}^G$. To combine these features, we propose a feature splicing strategy that captures both semantic and syntactic aspects. Specifically, we add a fully connected layer to linearly transform the fused features, and apply the $Sigmoid$ function to output the vulnerability prediction probability, thereby achieving the binary classification of SCVD.

$$\mathrm{X} = Concat(\hat{X}^C, \hat{X}^G, dim = 1) \tag{3}$$

$$Output = Sigmoid(\mathrm{X}) \tag{4}$$

III. Experimental Evaluation

A. Experimental settings

To evaluate MFVD, we conducted experiments aiming at addressing the following research questions:

RQ1: How does MFVD compare with state-of-the-art SCVD methods in terms of detection effectiveness and stability for reentrancy and timestamp dependency vulnerabilities?

RQ2: How effectively do different PLMs perform for the SCVD problem?

RQ3: How do different modules affect the detection performance?

Datasets. We selected two datasets: the SWD dataset [12], which contains 47,398 contracts labeled with 1,197 reentrancy vulnerabilities, and the ESC dataset [6], which contains 40,932 contracts labeled with 4,833 timestamp dependency vulnerabilities. We splited each dataset into training, validation, and test sets using an 80%-10%-10% ratio, and the evaluation metrics are precision, recall, and F1-score.

Parameter setup. We initialized the model with UniXcoder pre-trained weights [14]. The model configuration includes 6 layers with 12 attention heads each, and a maximum input length of 512 tokens. We used the AdamW optimizer with a learning rate of 2e-5. Batch sizes were set to 4 (training), 32 (validation), and 64 (testing). The model was trained for 5 epochs on SWD and 10 epochs on ESC, and we used the random seeds 3407, 42, 256, 1229, 2468, and 114514 in our experiments. All experiments were conducted on a system with an Intel I7-12700F CPU, 125 GB RAM, and an NVIDIA GeForce RTX 3080 Ti GPU (12 GB VRAM), running Ubuntu 22.04.4 LTS.

B. Experimental Results

1) Comparisons of baselines: Ror RQ1, we compared MFVD to 12 existing SCVD methods. Table II summarizes the results of multiple experimental runs on two types of vulnerabilities, where the performance is reported in terms of mean and variance. The results indicate that traditional methods perform poorly in both detection accuracy and stability, suggesting that these approaches struggle to effectively model vulnerable code as the complexity of contract data increase. Compared with existing deep learning–based methods, MFVD achieves higher detection accuracy on both vulnerability types while exhibiting stronger stability. Specifically, MFVD improves the F1 score over the state-of-the-art methods by 2.5%–4%. In addition, Table III reports the number of vulnerabilities detected in the test set, showing that MFVD is able to identify more real-world vulnerable contracts while maintaining a high F1 score.

2) Comparisons of PLMs: For RQ2, we compared 11 PLMs in Table IV with segmentation encoding strategy. UniXcoder performs well on this task, achieving F1 scores of 92.11% and 94.13% on the reentrancy and timestamp dependency vulnerability datasets, respectively. Other PLMs also demonstrate good performance on the vulnerability detection tasks, indicating the general applicability of pre-trained language models to code-related tasks and further confirming the stronger generalization ability of the model we selected.

3) Ablation experiments: To address RQ3, we conducted ablation experiments to evaluate the effectiveness of the two modules in MFVD. The experimental results are shown in Table V, where MFVD-WOG and MFVD-WOSC denote

TABLE II: Performance comparison of MFVD against 12 competitors regarding effectiveness and stability. "–" denotes not applicable.

Methods	Reentrancy			Timestamp Dependence		
	Recall (%)	Precision (%)	F1 (%)	Recall (%)	Precision (%)	F1 (%)
Mythril	62.41± 8.05	45.92± 5.48	51.84± 2.65	40.49± 0.88	50.07± 0.57	44.77± 0.74
Oyente	49.35± 3.61	60.84± 3.32	54.48± 3.74	40.43± 1.63	42.16± 2.43	41.20± 1.14
Securify	53.81± 2.65	53.89± 3.00	52.90± 1.00	–	–	–
Slither	61.32± 2.83	58.15± 2.24	59.70± 1.00	65.83± 1.04	67.03± 1.45	67.10± 1.79
Smartcheck	71.07± 2.45	76.98± 3.46	73.88± 1.41	36.77± 1.45	40.36± 1.05	38.48± 0.48
DR-GCN	81.28± 2.00	72.77± 1.73	76.75± 1.00	78.64± 0.71	71.82± **0.45**	75.07± 0.39
TMP	81.53± 1.73	75.14± 2.24	78.20± 2.00	84.55± 1.34	74.80± 0.65	79.38± 0.43
Peculiar	95.68± 3.16	88.88± 2.83	91.74± **0.28**	–	–	–
ReVulDL	93.37± 1.00	90.25± 1.41	92.32± 0.71	94.45± 1.05	92.07± 1.05	93.21± 0.36
EFEVD	87.28± 2.24	84.33± 1.73	86.74± 0.45	88.66± 0.45	87.45± 1.04	87.99± **0.11**
DMT	81.75± 1.41	83.18± 1.00	82.45± 0.63	95.12± 1.83	93.43± 1.05	94.26± 1.14
Clear	73.80± 4.47	81.18± 6.56	78.18± 6.63	64.39± 2.68	78.28± 2.68	70.58± 0.82
MFVD	**98.37± 0.22**	**94.12± 0.55**	**96.14±** 0.32	**97.46± 0.24**	**95.98±** 0.83	**96.71±** 0.44

TABLE III: Performance comparison regarding F1 and the number of detected vulnerabilities.

Methods	Reentrancy		Timestamp Dependence	
	F1(%)	Number	F1(%)	Number
Mythril	51.00	15	45.29	51
Oyente	59.30	21	41.53	43
Securify	53.76	102	–	–
Slither	58.74	88	69.72	174
Smartcheck	73.08	27	38.37	36
DR-GCN	76.39	174	75.42	177
TMP	80.40	189	79.52	193
Peculiar	91.83	218	–	–
ReVulDL	92.65	225	93.51	239
EFEVD	87.88	201	88.85	226
DMT	86.39	195	94.97	241
Clear	79.07	170	71.29	177
MFVD	**96.74**	**238**	**97.25**	**258**

TABLE IV: Performance comparison of 11 PLMs.

Models	Reentrancy			Timestamp Dependence		
	Recall (%)	Precision (%)	F1 (%)	Recall (%)	Precision (%)	F1 (%)
BERT [15]	50.00	49.70	49.85	65.85	86.87	71.52
RoBERTa [16]	50.00	46.53	48.20	51.05	76.61	50.34
LongCoder [22]	52.71	53.64	53.17	61.90	48.59	54.44
ELECTRA [18]	50.00	49.70	49.85	50.00	46.53	48.20
GraphCodeBERT [19]	88.67	**92.34**	90.32	88.15	94.78	91.15
CodeBERT [17]	89.17	91.59	90.34	90.98	93.83	92.34
CodeBERTa [17]	94.01	88.99	91.43	93.64	93.81	93.73
CodeT5 [20]	93.83	87.41	90.52	**94.63**	92.20	93.38
CodeT5+ [21]	89.45	92.83	91.06	92.82	93.17	92.29
PLBART [23]	61.81	87.43	67.89	93.25	95.10	**94.15**
UniXcoder	**96.04**	88.81	**92.11**	93.08	**95.25**	94.13

the variants that remove the graph processing module and the source code sequence processing module, respectively. Both variants suffer performance degradation of approximately 2%–4.5% in terms of F1 score. Notably, MFVD-WOG experiences a more significant performance drop than MFVD-WOSC, indicating that both source code sequences and graph structures play important roles in capturing vulnerability semantics. Furthermore, Table VI investigates the impact of removing the segmentation encoding strategy in the source code sequence processing module, referred to as MFVD-TSCS. The results show that the F1 score decreases by 6.22% and 6.52% for the two vulnerability detection tasks, respectively, demonstrating that the segmentation encoding strategy effectively captures the semantic features of the source code.

TABLE V: Results of ablation studies on two modules.

Modules	Reentrancy			Timestamp Dependence		
	Recall (%)	Precision (%)	F1 (%)	Recall (%)	Precision (%)	F1 (%)
MFVD-WOG	96.04	88.81	92.11	93.08	95.25	94.13
MFVD-WOSC	97.86	89.90	93.71	95.10	94.63	94.86
MFVD	**98.37**	**94.12**	**96.14**	**97.46**	**95.98**	**96.71**

TABLE VI: Results of ablation study on segmentation encoding.

Modules	Reentrancy			Timestamp Dependence		
	Recall (%)	Precision (%)	F1 (%)	Recall (%)	Precision (%)	F1 (%)
MFVD-TSCS	88.20	83.86	85.89	89.42	85.97	87.61
MFVD-WOG	**96.04**	**88.81**	**92.11**	**93.08**	**95.25**	**94.13**

IV. Conclusion

This paper investigates the limitations of existing smart contract vulnerability detection approaches and proposes a multimodal vulnerability detection framework, MFVD. The two modules in MFVD are designed to address issues about the code truncation and the loss of structural information, respectively, and experimental results fully demonstrate the effectiveness of the proposed framework. In addition, we conduct a systematic comparison of 11 PLMs, confirming their general applicability to code-related tasks.

V. Acknowledgement

This research was supported by the Scientific Research Fund of Hunan Provincial Education Department of China (24A0230) and the Natural Science Foundation of Hunan Province of China (2024JJ5043).

References

[1] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 254–269.

[2] 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.

[3] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[4] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[5] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, pp. 1133–1144, 2020.

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

[7] C. Jiang, X. Liu, S. Wang, J. Liu, and Y. Zhang, "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.

[8] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "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.

[9] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 378–389.

[10] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[11] P. Qian, Z. Liu, Y. Yin, and Q. He, "Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2220–2229.

[12] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 1349–1352.

[13] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[14] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, p. 7212–7225, 2022.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019, pp. 4171–4186.

[16] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *In Findings of the Association for Computational Linguistics*, pp. 1536–1547, 2020.

[18] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," in *ICLR*, 2020, pp. 1–18.

[19] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *In 9th International Conference on Learning Representations (ICLR)*, pp. 1–18, 2021.

[20] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *In the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1–13.

[21] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *The 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1–20.

[22] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: A long-range pre-trained language model for code completion," in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.

[23] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, 2021.