EFEVD: Enhanced Feature Extraction for Smart Contract Vulnerability Detection

Chi Jiang[1], **Xihan Liu**[1], **Shenao Wang**[1], **Jinzhuo Liu**[2], **Yin Zhang**[1*]
[1]University of Electronic Science and Technology of China
[2]Yunnan University
{chijiang, xihanliu, shenaowang}@std.uestc.edu.cn, jinzhuo.liu@hotmail.com, zhangyin123@uestc.edu.cn

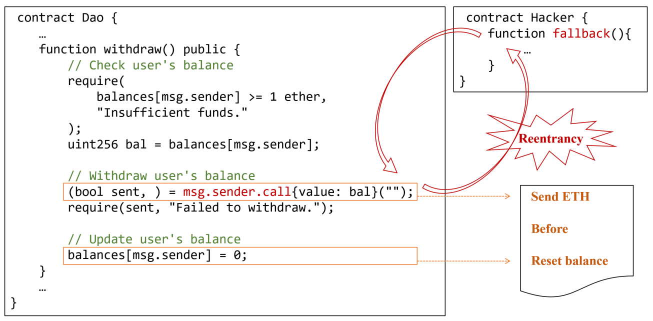
Abstract

Because of the wide deployment of smart contracts, smart contract vulnerabilities pose a challenging risk to blockchain security. Currently, deep learning-based vulnerability detection is a very attractive solution due to its ability to identify complex patterns and features. The existing methods mainly consider the contract code content features, expert knowledge patterns, and contract code modalities. To further enhance smart contract vulnerability detection, this paper attempts to identify community features from smart contracts with similar semantic and syntactic structures, and shared features from two related vulnerability detection tasks, vulnerability classification and localization. The experimental results verify that the proposed approach significantly outperforms the state-of-the-art methods in terms of accuracy, recall, precision, and F1-score.

1 Introduction

As a core component of blockchain technology, smart contracts significantly enhance the blockchain application potential through their ability to automatically execute transactions. However, this characteristic also creates substantial security risks. Once a smart contract is deployed, its code becomes immutable, meaning that any existing vulnerabilities can lead to irreversible losses. A prime example of this is the reentrancy vulnerability found in the DAO smart contract. As shown in Fig.1, in this vulnerability, the contract executed an external call (msg.sender.call) before updating its internal state (account balance), which allowed attackers to repeatedly trigger the withdrawal function and extract funds before the balance was updated. This demonstrates that contract vulnerabilities are intimately connected with code variables (semantics) and code structures (syntax). Building on this, Liu et al. [Liu *et al.*, 2021a; Liu *et al.*, 2021b] summarized expert knowledge patterns for reentrancy, timestamp, and infinite loop vulnerabilities, integrating expert pattern features with semantic and synthetic code analysis to enhance vulnerability detection performance.

[*]Corresponding Author



```
contract Dao {
    …
    function withdraw() public {
        // Check user's balance
        require(
            balances[msg.sender] >= 1 ether,
            "Insufficient funds."
        );
        uint256 bal = balances[msg.sender];

        // Withdraw user's balance
        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to withdraw.");

        // Update user's balance
        balances[msg.sender] = 0;
    }
    …
}
```

```
contract Hacker {
    function fallback(){
        …
    }
}
```

Figure 1: DAO smart contract with Reentrancy vulnerability

Subsequent works [Sendner *et al.*, 2023; Li *et al.*, 2023; Xie *et al.*, 2023] continued to explore expert patterns for additional vulnerability types, enhancing feature extractions. However, this expert pattern-based vulnerability detection approach, which relies on domain expertise, lacks generalizability. Another category of deep learning-based vulnerability detection methods focuses on the code's modality features[Qian *et al.*, 2023; Jie *et al.*, 2023]. These studies comprehensively consider the two primary modalities of smart contracts—the source code and EVM (ethereum virtual machine) bytecode—along with their respective features. However, methods based on code modality feature enhancement pose high demands on datasets, and are prone to introducing overlapping redundant features between different modalities, which can limit model performance.

Therefore, this paper considers two novel dimensions of feature enhancement for smart contract vulnerability detection. First, we introduce community features for smart contracts. In domains such as social networks, financial transactions, and academic citations, the related community specifies the common interests of the individuals, which are characterized by similar patterns and preferences[Hu *et al.*, 2023]. Given the close relationship between smart contract vulnerabilities and their semantic and syntactic properties, we define a community as a group of smart contracts with similar semantics and syntax. By analyzing patterns and vulnerabilities in similar contracts, we can extract a broader context, effectively aiding in the identification of the vulnerabilities. Second, we focus on the shared features related to two prevalent tasks in vulnerability detection, vulnerability classification and vulnerability localization. Vulnerability classification refers to deciding whether a smart contract contains vulnerabilities, while vulnerability localization involves classifying specific functions or variables within a contract for fine-grained vulnerability pinpointing. These cross-task shared features encompass not only the details of specific code segments but also the overall structure and behavioral patterns of the contracts[Xie *et al.*, 2020; Takiddin *et al.*, 2023], enabling the model to identify and locate vulnerabilities more effectively.

Specifically, this paper introduces an enhanced feature-based vulnerability detection (EFEVD) method for smart contracts. Initially, we employ the TextCNN and Transformer approaches for feature embedding of the internal function codes of smart contracts, and for extracting semantic features and sequence dependencies. Subsequently, a smart contract graph is constructed with functions as nodes and call relationships as edges. Using graph neural network (GNN) methods, the features encompassing the semantic and syntactic information of the contract graph are obtained. Thereafter, smart contracts are categorized into different communities based on the semantic and syntactic similarity between the contract graphs, and the commodity features are extracted. Finally, two downstream tasks for vulnerability detection are designed, classification of the contract graph and classification of the nodes within the graph. The model not only integrates the original features of the contract graph with the corresponding community features but also learns the cross-task shared features through the multitask learning architecture. This approach enhances the accuracy of the graph classification task while simultaneously locating specific vulnerable code segments through node classification.

Our main contributions can be summarized as follows:

- We explore the community features of smart contracts by defining communities based on semantic and syntactic similarities. Community features provide wider contextual information, enhancing the detection of vulnerabilities.
- We leverage shared features across two related tasks in vulnerability detection. By combining both detailed code segments and overarching contract analysis, our approach enhances the model's performance in identifying vulnerabilities.
- We employ a multitask learning architecture, learning and fusing enhanced features. We conduct extensive experiments on a real-world smart contract dataset, and our method outperforms the state-of-the-art methods, recall increases by 10.12%, F1-score increases by 8.17%, precision increases by 4%, and accuracy increases by 3.1%.

The remainder of this paper is organized as follows. Section 2 provides a comprehensive review of related work in the field of smart contract vulnerability detection. Section 3 formulates the proposed problem. Section 4 describes the proposed EFEVD method in detail. Section 5 presents the experimental information, including the dataset, evaluation metrics, and experimental results. Finally, Section 6 concludes the paper.

2 Related Work

Early works on vulnerability detection were primarily divided into two types, static analysis and dynamic analysis. Static analysis involves analyzing code without running the program, while dynamic analysis monitors the behavior of the code during its execution to detect vulnerabilities and errors. Chen et al[Chen *et al.*, 2021] developed a smart contract vulnerability detection tool called DefectChecker, which uses symbolic execution to detect eight types of vulnerabilities. Li et al[Li *et al.*, 2022] proposed a new tool called ReDefender, which uses fuzzy testing to detect reentrancy vulnerabilities. Osiris[Torres *et al.*, 2018], which used symbolic execution to verify potential integer overflow vulnerabilities in smart contract bytecode. Oyente[Luu *et al.*, 2016], which generates symbolic paths through symbolic execution to detect timestamp dependencies and reentrancy vulnerabilities; Smartcheck[Tikhomirov *et al.*, 2018], an Ethereum static analysis tool that uses an syntax parser and specifies Solidity syntax rules to detect various types of vulnerabilities. Contract Fuzzer[Jiang *et al.*, 2018], is a fuzz testing-based smart contract vulnerability detection framework.

However, static analysis often results in high false-positive rates and fails to capture runtime dynamics, while dynamic analysis is limited to executed code paths and is resource intensive. These drawbacks highlight the need for more effective strategies, which has led to an increasing focus on deep learning-based detection methods as promising alternatives.

Currently, deep learning-based methods in the blockchain domain predominantly leverage three categories of features. First, there are contract code content features. Ren et al.[Ren *et al.*, 2023] utilized complete semantic structure information (CPSs) to construct program slices, facilitating the learning of syntactic structures, keywords, and other pertinent information. Chen et al.[Chen *et al.*, 2023] introduced a novel semantic graph incorporating syntax and semantic information to represent the semantic details of each function comprehensively, enabling a more holistic capture of contract vulnerabilities. Liu et al.[Liu *et al.*, 2023] pioneered the construction of a Multi-Relationship Nested Contract Graph to better characterized the syntax and semantics of smart contract code. Second, expert knowledge pattern features are proposed. Liu et al.[Liu *et al.*, 2021b] designed specific expert patterns and integrated them with contract graph features, to construct a final vulnerability detection system. Based on this, an increasing number of vulnerability expert patterns are being progressively analyzed and proposed[Sendner *et al.*, 2023; Li *et al.*, 2023; Xie *et al.*, 2023]. Finally, the contract modality feature is considered. Qian et al.[Qian *et al.*, 2023] distinguished between two modalities: control flow graphs generated from bytecode and code semantic graphs generated from source code. Wang et al.[Wang *et al.*, 2023] proposed the DEEPVD model, which leverages three hierarchical modality features, including statement types, post-dominator trees, and exception flow graphs. These features are meticulously designed to enhance discriminative performance.

In summary, although existing works have considered some feature enhancements, there are still some limitations. Expert knowledge pattern features require predefined domain

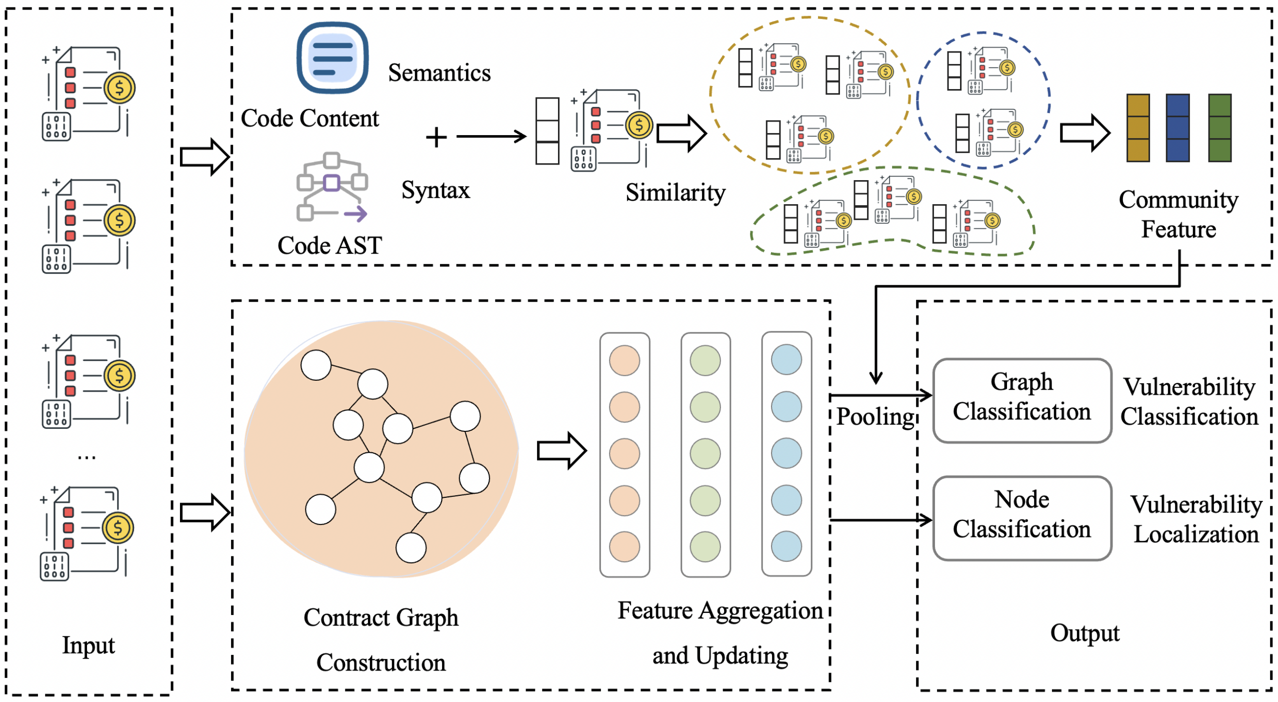


Figure 2: The overall architecture of the proposed method EFEVD

knowledge, while multi-modal features demand high-quality datasets and are prone to introducing redundant features. Therefore, we propose the extraction of two new types of readily obtainable smart contract features to enhance feature representation and improve detection performance.

3 Problem Formulation

In the enhanced feature extraction for smart contract vulnerability detection (EFEVD), we formalize the problem as follows:

Consider a smart contract graph g derived from the smart contract space G. Each contract graph g is composed of nodes N, representing contract functions, and edges representing call relationships between functions. We define two tasks within a multitask learning framework: graph classification ($\mathcal{GC}$) for overall contract vulnerabilities, and node classification ($\mathcal{NC}$) for function-specific vulnerabilities.

$$\mathcal{GC} : g \to [0, 1]$$
$$\mathcal{NC} : N \to [0, 1]$$

where $\mathcal{GC}(g)$ and $\mathcal{NC}(N)$ yield a probabilistic estimate of the presence of vulnerabilities in contract g and its functions N. The objective of the EFEVD is to minimize the cross-entropy between the predicted results and the true labels.

Let $\tilde{\mathcal{G}}$ represent community features. The shared features are refined through a multitask learning process (MTL):

$$g_{updated}, N_{updated} = \text{MTL}(g, N, \tilde{\mathcal{G}})$$
$$(g_{updated}, N_{updated}, \tilde{\mathcal{G}}) \to (\mathcal{GC}, \mathcal{NC})$$

This formulation reflects the community and shared feature enhancement of the EFEVD method, highlighting the integration of feature extraction techniques to enhance the vulnerability detection results of smart contracts.

4 Proposed Method

An overview of the proposed system is illustrated in Fig.2, which consists of three components, a community feature extraction module, a graph construction module, and a multitask downstream output module.

4.1 Community Feature Extraction

As mentioned before, we define a community in smart contracts as a group of contracts with similar semantic and syntactic features. Therefore, our first step is to extract the semantic and syntactic features of smart contracts. For the source code of smart contracts, the semantic features are embedded into vector space based on the word2vec method. The syntactic features are obtained by parsing the abstract syntax tree (AST) of the code, as shown in Fig.4. We classify smart contracts into different communities based on the similarity of their semantic and syntactic features. The similarity $Simi$ is defined as:

$$Simi = \frac{SemanSimi + SyntSimi}{2} \tag{1}$$

where $SemanSimi$ represents the semantic similarity between contracts, and $SyntSimi$ represents their syntactic similarity. Contracts with a similarity exceeding a certain threshold are classified into the same community. Additionally, to mine the feature embeddings corresponding to each

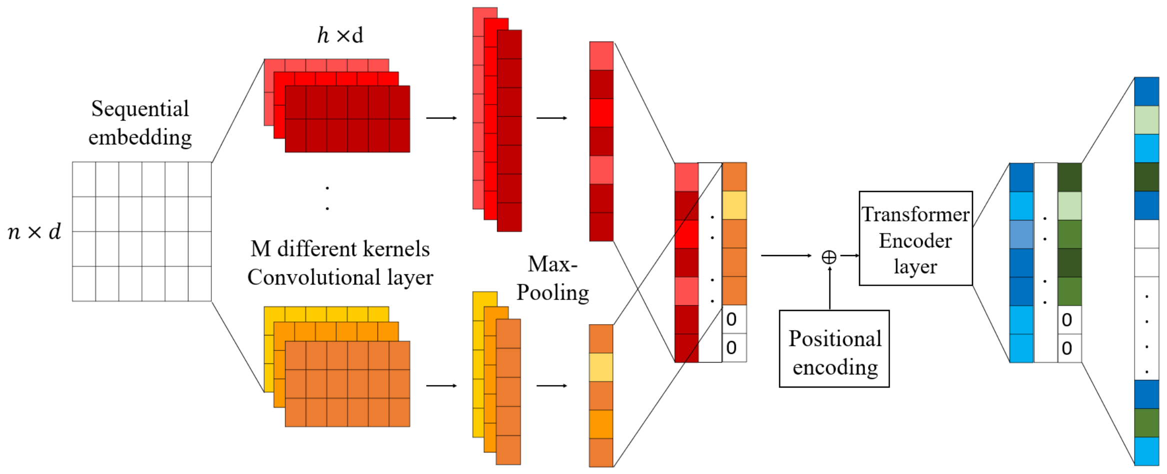
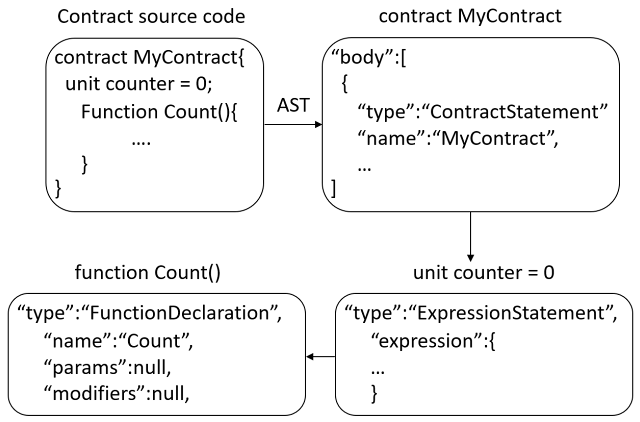


Figure 3: Function feature embedding procedure



```
contract MyContract{
  unit counter = 0;
    Function Count(){
          ....
    }
}
```

```
"body":[
 {
   "type":"ContractStatement"
   "name":"MyContract",
   ...
]
```

```
"type":"ExpressionStatement",
  "expression":{
  ...
  }
```

```
"type":"FunctionDeclaration",
   "name":"Count",
   "params":null,
   "modifiers":null,
```

Figure 4: An example of the abstract syntax tree of smart contract code

community, we establish edge connections between smart contracts within the same community. If the similarity between two smart contracts exceeds 80%, an edge is created between their respective nodes. This process transforms each community into a graph form, and then, based on GNN methods, we extract the feature representations for each type of community, defined as $\tilde{\mathcal{G}} = \{\tilde{G_1}, \tilde{G_2}, \dots \tilde{G_n}\}$

4.2 Contract Graph Construction

For each smart contract, a smart contract graph is constructed with its functions as nodes and the invocation relationships between the functions as edges. To extract the node feature, each function is embedded into the features as a sequence. First, word2vec is used to generate word embeddings for each word in the function. Then, the TextCNN and Transformer are used to generate feature embedding for the function, better capturing short and long term sequence features.

Formally, after word2vec embedding, each function is represented as $x = [x_1, x_2, ..., x_n] \in R^{n \times d}$, where n is the sequence length and d is the dimension of the word vector. The TextCNN feature extractor can slide different sizes of convolutional kernels over the input sequence to aggregate local features of different lengths. Assuming M filters with R convolution kernels of size $h \times d$ are used, the feature vector obtained by the $m-th$ filter is represented as:

$$c_m = max((f(W^m x_{i:i+h_m-1} + b))_{i=1}^{n-h_m+1}) \qquad (2)$$

Here, $W^m \in R^{h_m \times d}$ represents the weight matrix of the $m-th$ filter in the convolution operation, b is the bias, $f(\cdot)$ denotes the activation function, and $max(\cdot)$ represents max pooling. Finally, all feature vectors of the filters are concatenated into a complete sequence $C = [c_1, c_2, ..., c_m]$ as the input to the Transformer.

The Transformer encoding layer consists of a multi-head Attention mechanism and a feedforward neural network. The self-attention layer computes attention weights between input embedding vectors to represent the importance of each input vector in the sequence. The multi-head attention layers and feedforward networks further process these encoded vectors to obtain more semantically meaningful textual representations, as shown below:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \qquad (3)$$

Here,Q, K, V represent the query, key, and value vectors, respectively, with d_k being the vector dimension. In our model, we first utilize TextCNN with various kernel sizes to capture multi-scale features, defined as C. C is then fed into Transformer to learn inter-scale relationships. In Transformer, C is first transformed into three different sets of vectors: query ($Q = CW^Q$), key ($K = CW^K$), and value ($V = CW^V$), where W^Q, W^Q, and W^V are learnable weight matrix. The resulting Q, K, and V vectors are then used in the computation of attention scores and the subsequent generation of the output sequence $\tilde{C}$. The new sequence representation $\tilde{C}$ generated by multi-head self-attention on the input sequence C is processed through a feedforward neural network, where a non-linear transformation is applied to each token in the sequence. The function feature embedding is denoted as E_f:

$$E_f = FFN(\hat{C}) = max(0, \hat{C}W_1 + b_1)W_2 + b_2 \tag{4}$$

where W_1, b_1, W_2 and b_2 are the learnable weight matrices and bias vectors. The overall process of function feature embedding is illustrated in Fig.3.

After obtaining the node feature in the contract graph, the node feature is propagated according to the order of edges based on GNN. At time step t, node information is transmitted through edge e_k to its end node. Specifically, the message m_k is computed based on the hidden state of the start node of e_k, and the edge embedding e_k:

$$x_k = h_{e_k^{start}} \oplus e_k \tag{5}$$

$$m_k = W_k x_k + b_k \tag{6}$$

where $\oplus$ denotes the concatenation operation, and W_k and b_k are network parameters. Upon receiving the message feature, the end node of the edge updates its current hidden state h_{ek} by considering the previous hidden state and the edge-specific update m_k:

$$\hat{h}_{ek} = tanh(Um_k + Zh_{ek} + b_3) \tag{7}$$

$$h_{ek}^{'} = softmax(R\hat{h}_{ek} + b_4) \tag{8}$$

where U, Z and R are matrices, b_3 and b_4 are bias vectors. Finally, for the smart contract graph with $|V|$ nodes, the overall feature embedding of the smart contract obtained by aggregating the node features is:

$$E_c = \sum_{i=1}^{|V|} h_i^{'} \tag{9}$$

4.3 Multi-task Downstream Output

In the final output layer, we defined two tasks related to vulnerability detection: vulnerability classification and vulnerability localization, corresponding to the classification of the smart contract graph and the classification of individual graph nodes, respectively. In the process of multi-task learning, information pertinent to the tasks is interchanged and enriched through a shared feature at an early stage, leading to simultaneous refinement of both node and graph classification tasks. For the contract graph classification task, the input is the fusion of contract graph feature E_c from the multi-task learning process and contract community feature $\tilde{\mathcal{G}}$. The output of the classifier is defined as $\hat{y}^1$, and y^1 is the true label of the sample. The cross-entropy loss function is used to train the model:

$$Loss_{\mathcal{GC}} = -y^1 log\hat{y}^1 - (1 - y^1)log(1 - \hat{y}^1) \tag{10}$$

The loss of node classification $Loss_{\mathcal{NC}}$ is similar to Equation(10). We achieve multi-task learning on graphs by designing a joint loss function that combines the two masked categorical cross-entropy losses for graph classification and node classification:

$$Loss = Loss_{\mathcal{GC}} + Loss_{\mathcal{NC}} \tag{11}$$

5 Experiments

In this section, we conduct experiments on a real-world smart contract dataset. We endeavor to investigate the following research questions:

RQ1: How does the proposed EFEVD method perform compared with traditional detection methods and state-of-the-art deep learning based methods?

RQ2: How do the proposed enhanced features in EFEVD perform with each other to improve the performance?

RQ3: How do hyper-parameters impact performance?

5.1 Experimental Settings

Datasets. The dataset used in this paper comes from a real-world smart contract dataset on Ethereum. The labels in the smart contract dataset are binary, with 1 indicating the existence of vulnerabilities and 0 indicating the absence of vulnerabilities. The dataset mainly contains four types of vulnerabilities, namely reentrancy, timestamp, overflow, and secure(No Vulnerability). The labels for the functions were obtained using the Slither tool combined with manual verification. After filtering out samples with development language types that do not match the vulnerabilities, the original dataset is preprocessed and balanced, resulting in a total of 9742 smart contract data samples. The overall dataset is divided into a train set and a test set at a ratio of 7:3, where the train set is used for model training and the test set is used for model performance evaluations.

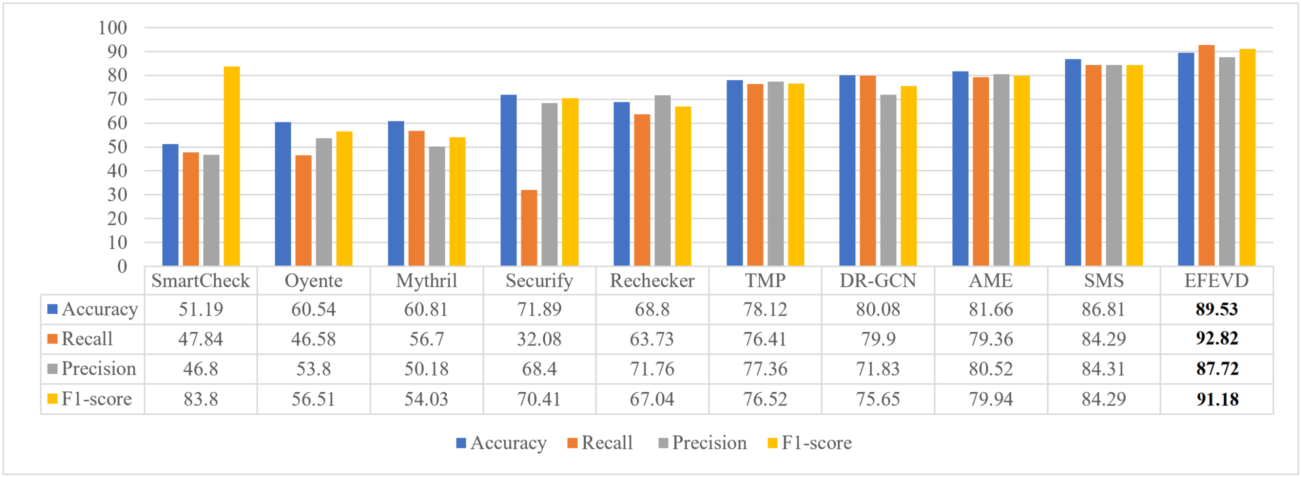
Compared Methods. To evaluate the improvement of the model's performance compared to that of existing methods, this paper adopts several highly recognized smart contract vulnerability detection methods as comparative experimental methods. The baselines can be categorized into four aspects:

- Static and dynamic analysis: This category includes tools such as SmartCheck [Tikhomirov *et al.*, 2018], which uses static analysis to detect common vulnerabilities. Oyente [Liu *et al.*, 2018], on the other hand, is a symbolic execution tool for detecting security flaws. Mythril [Feist *et al.*, 2019] combines concolic analysis and taint analysis, while Securify [Tsankov *et al.*, 2018] identifies vulnerabilities by analyzing the data flow in contracts.

- Semantics and syntax feature-based methods: These methods leverage the intrinsic features of smart contracts to detect vulnerabilities. Rechecker [Qian *et al.*, 2020] analyzes the semantic information to identify hidden bugs. TMP [Zhuang *et al.*, 2020] and DR-GCN [Zhuang *et al.*, 2020] employs deep graph networks to analyze the syntactic structure of smart contracts.

- Expert knowledge enhanced method: The AME [Liu *et al.*, 2021a] system incorporates expert knowledge into the detection process, combining the insights of smart contract security experts.

- Multi-modal feature enhancement method: SMS [Qian *et al.*, 2023] integrates features from two different modalities: bytecode and source code.



	SmartCheck	Oyente	Mythril	Securify	Rechecker	TMP	DR-GCN	AME	SMS	EFEVD
■ Accuracy	51.19	60.54	60.81	71.89	68.8	78.12	80.08	81.66	86.81	**89.53**
■ Recall	47.84	46.58	56.7	32.08	63.73	76.41	79.9	79.36	84.29	**92.82**
■ Precision	46.8	53.8	50.18	68.4	71.76	77.36	71.83	80.52	84.31	**87.72**
■ F1-score	83.8	56.51	54.03	70.41	67.04	76.52	75.65	79.94	84.29	**91.18**

Figure 5: Comparison experimental results of smart contract vulnerability detection

Implementation details. All the models are trained by the Adam optimizer. The loss function is the softmax cross-entropy loss, which is consistent with the underlying model architecture. The batch size is set to 64, and the learning rate is initialized to 0.01 and is dynamic during training. All the models are trained using an NVIDIA GeForce RTX 3060 GPU and implemented with TensorFlow 1.15. Our data and code are available on GitHub[1].

Metrics. This paper uses four evaluation metrics, accuracy, recall, precision, and F1-score, to validate the experimental results. The accuracy represents the ability to correctly identify each class. Recall represents the proportion of samples with existing vulnerabilities that are correctly predicted as having vulnerabilities, which is mainly used to measure the probability of missed detection. Precision measures the proportion of samples predicted as the positive class that are actually positive. The F1-score is the harmonic mean of precision and recall, providing a balanced consideration of the model's accuracy and coverage.

5.2 Overall Performance(RQ1)

The experimental results of the comparison of smart contract vulnerability classification tasks are shown in Fig.5. Overall, the deep learning-based detection methods are generally better than the traditional detection tools, and their accuracy can be maintained at approximately 80%. The recall results of traditional methods is less than 60%, indicating that they miss a substantial number of true vulnerabilities. Within the deep learning-based approaches, the DR-GCN and TMP methods consider only the semantic and syntactic features of code, resulting in relatively poor vulnerability detection outcomes. Building upon this, the AME method integrates expert knowledge features of code, and the SMS method incorporates multimodal features of code, both demonstrating improved detection results over the DR-GCN and TMP methods, which substantiates the efficacy of feature enhancements. However, neither of these feature enhancement approaches achieves the effectiveness of the proposed EFEVD method. EFEVD outperforms the existing state-of-the-art methods, achieving a 10.12% increase in recall, an 8.17% increase in F1-score, a 4% increase in precision, and a 3.1% increase in accuracy, demonstrating the effectiveness of the two types of enhanced features considered in our proposed method.

[1]https://github.com/xawmx/MTLContractdetection

5.3 Ablation Study(RQ2)

To discover the impact of our community- and shared- enhanced features, as well as the influence of several deep learning feature extraction methods used in our model on the vulnerability detection results, we conduct an ablation analysis by altering the following components:

- EFEVD(tt-): Without employing TextCNN and Transformer for semantic feature extraction, instead exclusively utilizing Word2Vec embedding along with Transformer positional encoding.
- EFEVD(sl-): Remove the syntax similarity during dividing the smart contracts communities.
- EFEVD(cf-): Remove the community feature enhancement. Only use the contract graph feature for vulnerability classification.
- EFEVD(sf-): Remove the shared feature enhancement. That is, remove the multi-task framework. Only consider the vulnerability classification task.

Ablation studies confirm the effectiveness of the proposed enhanced feature method. As shown in Table 1, replacing certain feature embedding components in the model with simpler methods, or removing either community features or shared features, leads to a decrease in performance across all four evaluation metrics. Specifically, The removal of TextCNN models in EFEVD(tt-) led to a decrease in all the metrics, emphasizing the importance of multi-scale semantic processing for vulnerability detection. Omitting syntax similarity in EFEVD(sl-) also led to a moderately reduced performance, highlighting syntax's role in accurately dividing smart contract communities. The exclusion of community feature in

Model	Vulnerability Classification			
	Accuracy	Recall	Precision	F1-score
EFEVD(tt-)	85.22	90.74	80.33	85.22
EFEVD(sl-)	86.96	89.51	83.82	86.57
EFEVD(cf-)	87.25	88.8	84.71	86.75
EFEVD(sf-)	88.41	89.08	84.27	88.24
EFEVD	**89.53**	**92.82**	**87.72**	**91.18**

Table 1: Ablation experiment results

EFEVD(cf-) and shared feature in EFEVD(sf-) negatively affected all performance metrics, thereby confirming the effectiveness of the two types of enhanced features we proposed for detecting vulnerabilities. Overall, the complete EFEVD model outperformed all other variants, further validating the effectiveness of the enhanced features and their combination.

5.4 Hyper-Parameters Sensitivity(RQ3)

The hyperparameters, such as the learning rate and number of epochs, have a direct impact on the speed and performance of the model. The number of epochs refers to the number of times the model is trained on the data, and an appropriate number of epochs can ensure that the model stops training after having sufficient fitting ability, thus reducing the training cost. In this section, we set four initial values of the learning rate, namely 0.1, 0.01, 0.001, and 0.0001, and record the convergence performance of the model under different conditions. To comprehensively evaluate the precision and recall performance, we still use the F1-score as the evaluation metric, and the results of the parameter experiment are shown in Fig.6. Both of the vulnerability detection tasks improve when the learning rate is set between 0.01 and 0.001. However, when the learning rate is set to 0.0001, the evaluation metric tends to decrease. Considering all aspects, we choose 0.01 as the initial learning rate and adjust the learning rate during model training by $\frac{0.01}{num_i ter}$, where $num_i ter$ represents the number of training groups in which 10 epochs are grouped. This helps the model to quickly find the global optimum. For the training epochs used for model training, we set the number of epochs to range from 1 to 100, and gradually increase it to observe the convergence of the loss and accuracy during model training. The parameter experimental results are shown in Fig.7. Based on the loss curve, the model's learning effect is very obvious in the first 30 training epochs, and the loss value shows a significant downward trend. When the epoch is set to 30-50, the model tends to converge, and when the epoch is set to 50-100, the model converges. Therefore, the epoch hyperparameter in this paper is set to 30.

6 Conclusion

In conclusion, the proposed EFEVD method introduces novel feature enhancement for smart contract vulnerability detection. This approach diverges from the existing domain-expertise-reliant methods by incorporating community features, which reflect common patterns that are beneficial for identifying vulnerabilities. Moreover, EFEVD leverages a multi-task learning framework that utilizes shared features across two related tasks, enhancing the model's performance. The experimental results indicate that EFEVD achieves superior performances over the current state-of-the-art methods, demonstrating its potential as an effective tool for improving smart contract security.

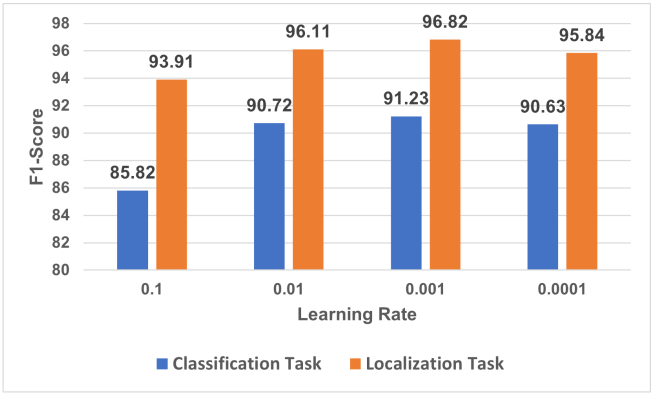


Figure 6: Experimental results of learning rate parameters

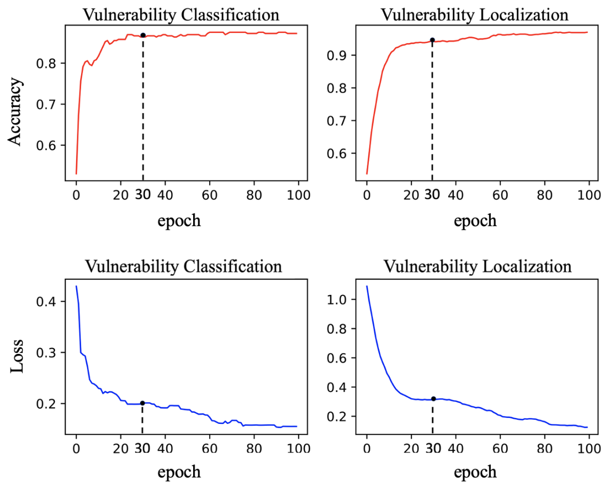


Figure 7: Experimental results of epoch selection

Acknowledgements

This work was supported by the China National Science Foundation under grants 62172079 and U23A20310.

References

[Chen *et al.*, 2021] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering*, 48(7):2189–2207, 2021.

[Chen *et al.*, 2023] Da Chen, Lin Feng, Yuqi Fan, Siyuan Shang, and Zhenchun Wei. Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention. *Journal of Systems and Software*, 202:111705, 2023.

[Feist *et al.*, 2019] Josselin Feist, Gustavo Grieco, and Alex 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)*, pages 8–15. IEEE, 2019.

[Hu *et al.*, 2023] Qifu Hu, Ruyang Li, Qi Deng, Yaqian Zhao, and Rengang Li. Enhancing network by reinforcement learning and neural confined local search. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, pages 2122–2132, 2023.

[Jiang *et al.*, 2018] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[Jie *et al.*, 2023] Wanqing Jie, Qi Chen, Jiaqi Wang, Arthur Sandor Voundi Koe, Jin Li, Pengfei Huang, Yaqi Wu, and Yin Wang. A novel extended multimodal ai framework towards vulnerability detection in smart contracts. *Information Sciences*, 636:118907, 2023.

[Li *et al.*, 2022] Bixin Li, Zhenyu Pan, and Tianyuan Hu. Redefender: detecting reentrancy vulnerabilities in smart contracts automatically. *IEEE Transactions on Reliability*, 71(2):984–999, 2022.

[Li *et al.*, 2023] Zhaoxuan Li, Siqi Lu, Rui Zhang, Ziming Zhao, Rujin Liang, Rui Xue, Wenhao Li, Fan Zhang, and Sheng Gao. Vulhunter: Hunting vulnerable smart contracts at evm bytecode-level via multiple instance learning. *IEEE Transactions on Software Engineering*, 2023.

[Liu *et al.*, 2018] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68, 2018.

[Liu *et al.*, 2021a] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint arXiv:2106.09282*, 2021.

[Liu *et al.*, 2021b] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[Liu *et al.*, 2023] Haiyang Liu, Yuqi Fan, Lin Feng, and Zhenchun Wei. Vulnerable smart contract function locating based on multi-relational nested graph convolutional network. *Journal of Systems and Software*, page 111775, 2023.

[Luu *et al.*, 2016] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[Qian *et al.*, 2020] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, 8:19685–19695, 2020.

[Qian *et al.*, 2023] Peng Qian, Zhenguang Liu, Yifang Yin, and Qinming He. Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In *Proceedings of the ACM Web Conference 2023*, pages 2220–2229, 2023.

[Ren *et al.*, 2023] Xiaojun Ren, Yongtang Wu, Jiaqing Li, Dongmin Hao, and Muhammad Alam. Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network. *Computers and Electrical Engineering*, 109:108766, 2023.

[Sendner *et al.*, 2023] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In *NDSS*, 2023.

[Takiddin *et al.*, 2023] Abdulrahman Takiddin, Rachad Atat, Muhammad Ismail, Katherine Davis, and Erchin Serpedin. A graph neural network multi-task learning-based approach for detection and localization of cyberattacks in smart grids. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2023.

[Tikhomirov *et al.*, 2018] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pages 9–16, 2018.

[Torres *et al.*, 2018] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.

[Tsankov *et al.*, 2018] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[Wang *et al.*, 2023] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. Deepvd: Toward class-separation features for neural network vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2249–2261. IEEE, 2023.

[Xie *et al.*, 2020] Yu Xie, Maoguo Gong, Yuan Gao, AK Qin, and Xiaolong Fan. A multi-task representation learning architecture for enhanced graph classification. *Frontiers in neuroscience*, 13:1395, 2020.

[Xie *et al.*, 2023] Xueshuo Xie, Haolong Wang, Zhaolong Jian, Yaozheng Fang, Zichun Wang, and Tao Li. Block-gram: Mining knowledgeable features for efficiently smart contract vulnerability detection. *Digital Communications and Networks*, 2023.

[Zhuang *et al.*, 2020] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural network. In *IJCAI*, pages 3283–3290, 2020.