# The Best of Both Worlds: Integrating Semantic Features with Expert Features for Smart Contract Vulnerability Detection

Xingwei Lin[1], Mingxuan Zhou[2], Sicong Cao[2(✉)], Jiashui Wang[1,3], and Xiaobing Sun[2]

[1] Ant Group, Hangzhou 310000, China
xwlin.roy@gmail.com, jiashui.wjs@antgroup.com
[2] College of Information Engineering, Yangzhou University, Yangzhou 225009, China
MZ120220958@stu.yzu.edu.cn, {DX120210088,xbsun}@yzu.edu.cn
[3] Polytechnic Institute, Zhejiang University, Hangzhou 310015, China

**Abstract.** Over the past few years, smart contract suffers from serious security threats of vulnerabilities, resulting in enormous economic losses. What's worse, due to the immutable and irreversible features, vulnerable smart contracts which have been deployed in the the blockchain can only be detected rather than fixed. Conventional approaches heavily rely on hand-crafted vulnerability rules, which is time-consuming and difficult to cover all the cases. Recent deep learning approaches alleviate this issue but fail to explore the integration of them together to boost the smart contract vulnerability detection yet. Therefore, we propose to build a novel model, SmartFuSE, for the smart contract vulnerability detection by leveraging the best of semantic features and expert features. SmartFuSE performs static analysis to respectively extract vulnerability-specific expert patterns and joint graph structures at the function-level to frame the rich program semantics of vulnerable code, and leverages a novel graph neural network with the hybrid attention pooling layer to focus on critical vulnerability features. To evaluate the effectiveness of our proposed SmartFuSE, we conducted extensive experiments on 40k contracts in two benchmarks. The experimental results demonstrate that SmartFuSE can significantly outperform state-of-the-art analysis-based and DL-based detectors.

**Keywords:** Smart contract · Vulnerability detection · Code representation Learning · Graph neural network · Expert features

## 1 Introduction

Smart contracts are programs or transaction protocols which automatically execute on the blockchain [21]. The decentralization and trustworthy properties of smart contract have attracted considerable attention from different industries,


J. Chen et al. (Eds.): BlockSys 2023, CCIS 1897, pp. 17–31, 2024.
https://doi.org/10.1007/978-981-99-8104-5_2

and are used to support many tasks such as access control, task management or data management [9,22,35]. Taking the most famous blockchain platform Ethereum [31] as an example, there are more than 1.5 million smart contracts have been deployed [23].

However, due to several properties [29], smart contract is more vulnerable to attacks than traditional software programs [5,36]. On the one hand, the transparency of smart contracts expose a large attack surface to hackers, allowing them to call a smart contract with no limitations. On the other hand, as the blockchain is immutable and irreversible, once a vulnerable smart contract is deployed on the blockchain, it neither can be repaired nor interrupted. Considering the serious impact of smart contract vulnerabilities [6], timely detection of smart contract vulnerabilities is necessary and urgent.

Conventional smart contract vulnerability detection approaches often adopt static or dynamic analysis techniques. Unfortunately, these approaches fundamentally rely on several fixed expert rules, while the manually defined patterns bear the inherent risk of being error-prone and some complex patterns are non-trivial to be covered. Recently, benefiting from the powerful performance of Deep Learning (DL), a number of approaches [3,24,25,37] have been proposed to leverage DL models to learn program semantics to identify potential vulnerabilities. However, existing DL-based detection approaches fail to precisely model and extract critical features related to vulnerabilities, leading to unsatisfactory results, i.e., either missing vulnerabilities or giving overwhelmingly false positives.

To cope with the aforementioned challenges, in this paper, we propose a novel approach, called SmartFuSE, which captures the distinguishing features of Smart contract vulnerabilities by Fusing Semantic features with Expert features. In particular, SmartFuSE firstly performs static analysis to respectively extract vulnerability-specific expert patterns and joint graph structures at the function-level to frame the rich program semantics of vulnerable code. Considering that even a single function could have hundreds lines of code, which may introduce much noise, SmartFuSE performs forward and backward slicing from the program point of interest based on control- and data-dependence to extract vulnerability-related code snippets. Second, we leverage Gated Graph Neural Network (GGNN) with hybrid attention pooling layer to focus on critical vulnerability features and suppress unimportant ones via the attention mechanism. Finally, the local expert features and global semantic features are fused to produce the final vulnerability detection results. To evaluate the effectiveness of our proposed SmartFuSE, we conducted extensive experiments on 40k contracts in two benchmarks. The experimental results demonstrate that SmartFuSE can significantly outperform state-of-the-art analysis-based and DL-based detectors.

The main contributions can be summarized as follows:

– We propose to characterize the contract function source code as graph representations. To focus on vulnerability-related features, we employ several expert patterns and program slicing to capture local and global vulnerability semantics.

– We propose a novel DL-based smart contract vulnerability detection technique, SmartFuSE, with a fusion model to extract the distinguishing features from global semantic features and local expert features of source code.
– We comprehensively investigate the value of integrating the semantic features and expert features for smart contract vulnerability detection. The results indicate that SmartFuSE outperforms the state-of-the-art approaches.

## 2 Background

### 2.1 Smart Contract Vulnerabilities

In this work, we concentrate on the following three common types of vulnerabilities in smart contract.

**Reentrancy** vulnerability occurs when the caller contract is simultaneously entered twice. In traditional programs, the execution is atomic when called a non-recursive function and there will be no new function execution before the current function execution ends. However, in smart contract, the malicious callee external contract may reenter the caller before the caller contract finishes when conducting an external function call.

**Timestamp dependence** vulnerability happens when *block.timestamp* is leveraged to trigger certain critical operations, e.g., generating specific numbers. Since the miners in the blockchain has the freedom to adjust the timestamp of the block as long as it is within a short time interval, they may manipulate the block timestamps to gain illegal benefits.

**Infinite loop** vulnerability, which unintentionally iterates forever, occurs when a smart contract contains a loop statement with no (or unreachable) exit condition. Such vulnerability will consume a lot of gas but all the gas is consumed in vain since the execution is unable to change any state.

### 2.2 Graph Neural Networks

Due to the outstanding ability in learning program semantics, Graph Neural Networks (GNNs) have been applied to a variety of security-related tasks achieved great breakthroughs. Modern GNNs follow a neighborhood aggregation scheme, where the representation of a node is updated by iteratively aggregating representations of its $k$-hop neighbors, to capture the structural information of graphs. This procedure can be formulated by:

$$\boldsymbol{h}_v^{(t)} = \sigma\left(\boldsymbol{h}_v^{(t-1)}, AGG^{(t)}\left(\left\{\boldsymbol{h}_u^{(t-1)} : u \in \mathcal{N}(v)\right\}\right)\right) \tag{1}$$

where $\boldsymbol{h}_v^{(t)}$ is the feature representation of node $v$ at the $t$-th iteration, $u \in \mathcal{N}(v)$ is the neighbors of $v$, and $AGG(\cdot)$ and $\sigma(\cdot)$ denote aggregation (e.g., $MEAN$) and activation (e.g., $ReLU$) functions for node feature computation.

According to different goals, the final node representation $h_v^{(T)}$ can be used for graph classification, node classification, and link prediction [33].

**Graph Classification.** Given a graph $G_i = (V, E, X) \in \mathcal{G}$ and a set of graph labels $\mathcal{L} = \{l_1, \cdots, l_m\}$, where each node $v \in V$ is represented by a real-valued feature vector $\boldsymbol{x}_v \in X$ and $m$ denotes the number of graph labels, graph classification aims to learn a mapping function $f : \mathcal{G} \rightarrow \mathcal{L}$ to predict the label of the $i$-th graph $G_i$.

**Node Classification.** Given a graph $G_j = (V, E, X) \in \mathcal{G}$ and its node label set $\mathcal{L} = \{l_1, \cdots, l_n\}$, node classification aims to learn a mapping function $g : \mathcal{V} \rightarrow \mathcal{L}$ to predict the label of node $v$.

**Link Prediction.** Given node $u$ and node $v$, link prediction aims to predict the probability of connection between node $u$ and node $v$ by $y_{u,v} = \phi\left(\boldsymbol{h}_u^{(k)}, \boldsymbol{h}_v^{(k)}\right)$, where $\boldsymbol{h}_u^{(k)}$ and $\boldsymbol{h}_v^{(k)}$ are the node representations after $k$ iterations of aggregation and $\phi(\cdot)$ refers to the composition operator such as *Inner Production*.

Considering that existing GNNs suffer from the long-term dependency issue, which prevents nodes from effectively transferring messages, our work builds on Gated Graph Neural Network (GGNN) [15]. GGNN uses a gated recurrent unit to remember the key features. This allows GGNN to go deeper than other GNNs. So this deeper model has a powerful ability to learn more semantics features from graph data.

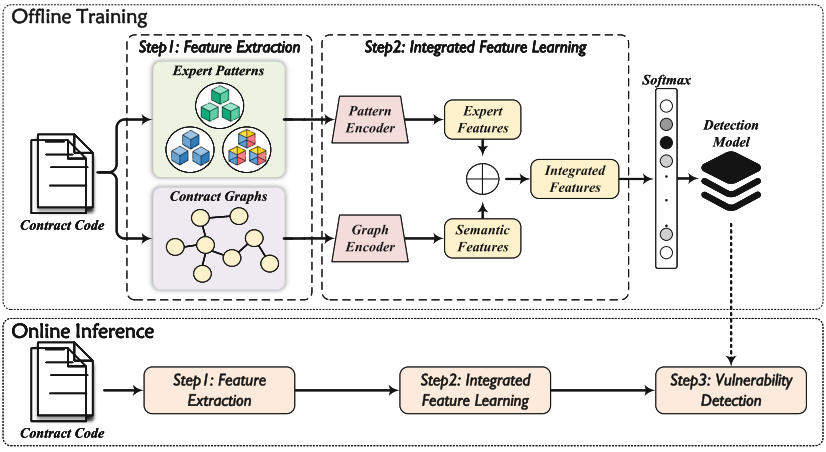


**Fig. 1.** The overall workflow of our approach.

## 3 Our Approach

The overview of our proposed approach is illustrated in Fig. 1, which consists of three components: 1) feature extraction, where vulnerability-specific expert patterns are obtained by adopting a fully automatic tool proposed by Liu et al.

[17] and semantic features are extracted with GGNN with a hybrid attention pooling layer; 2) integrated feature learning is proceeded with a fully connected layer, and 3) vulnerability detection is to predict the vulnerable smart contracts at the function-level with previously learned features. Details of SmartFuSE are presented in the following subsections.

### 3.1 Feature Extraction

Feature extraction aims at converting the statistical data (i.e., expert features) and code representations (i.e., semantic features) into numeric representation that can be adapted to the deep neural network models to capture the distinguishing characteristics for the later vulnerability detection tasks.

**Expert Feature Extraction.** Expert features are a set of descriptive rules defined by experts according to their understanding of security-critical programming patterns with their professional knowledge and experience. In the literature, dozens of expert features have been defined from various dimensions (e. g., vulnerability types, security operations). The nine features (presented in Table 1) defined by Liu et al. [17] specific to three common smart contract vulnerabilities have been demonstrated for their effectiveness. Therefore, we follow the state-of-the-art work to adopt the nine expert features as an important part to mine the integrated features of source code for smart contract vulnerability detection.

**Table 1.** Studied nine common vulnerability-specific expert features.

| Vulnerability Type | Expert pattern | Security-Critical Operations |
|---|---|---|
| Reentrancy | enoughBalance | call.value invocation |
| | callValueInvocation | a function that contains call.value |
| | balanceDeduction | the variable: correspond to user balance |
| Timestamp dependence | timestampInvocation | block.timestamp invocation |
| | timestampAssign | block.number invocation |
| | timestampContaminate | a variable: affect critical operation |
| Infinite loop | loopStatement | for |
| | loopCondition | while |
| | selfInvocation | self-call function |

**Semantic Feature Extraction.** Semantic features represent the theoretical units of meaning-holding components which are used for representing word meaning, and capture the meaning of tokens in code as well as their contexts(e.g., semantic and syntactic structural information of code), that have been widely used to represent the intrinsic characteristics of code mined with DL models. In this work, we adopt a widely-used abstract representation Code Property Graph (CPG) [34], which merges Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG) into a joint data structure, to

reserve sufficient syntax and semantic features of source code. In addition, considering that a single function usually contains dozens of code lines while the vulnerability exists only in one or several lines of code, we further perform backward and forward program slicing [30] based on CPG from a program point of interest to filter noise induced by irrelevant statements. Our slice criteria can be divided into two categories: sensitive operations and sensitive data. For example, sensitive operations such as *.delegatecall* or *.call* will call an external contract may which may be malicious, while sensitive data are the insecure data which hackers can manipulate (e.g., *block.number*, *block.timestamp*). The details of our slice criteria are listed in Table 2.

**Table 2.** Studied eight common vulnerability-specific expert features.

| Slice Criteria | Example |
|---|---|
| block info | *block.number*, *block.gaslimit* |
| delegatecall | *address.delegatecall* |
| arithmetic operations | $*, +, -$ |
| external function call | *address.call* |
| selfdestruct operation | *selfdestruct* |
| input address parameters | *address input_ address* |
| block timestamp | *block.timestamp* |
| low level call operation | *address.call* |

### 3.2 Integrated Feature Learning

With the expert and semantic features extracted in different ways, SmartFuSE further needs to encode them into feature vectors and learn their integrated features. For local expert features, we adopt a feed-forward neural network (FNN) as feature encoders. For global semantic features, we utilize GGNN with a hybrid attention pooling layer to transform sliced contract graphs into deeper graph features. Then, the expert features and semantic features are merged by using a fusion network for vulnerability detection.

**Local Expert Feature Learning.** Each expert pattern formulates an elementary factor closely related to a specific vulnerability. We utilize One-Hot Encoding to represent each pattern, i.e., "0"/"1" indicates whether the function under test has this pattern or not, respectively. The vectors for all patterns related to a specific vulnerability are concatenated into a final vector $x$. The final vector $x$ and the ground truth of whether the function has the specific vulnerability as the target label will be fed into a feed-forward neural network $\psi(\cdot)$ with a convolution layer and a max pooling layer to extract its corresponding $d$-dimensional expert feature $E_r$. The convolutional layer learns to assign different weights to

different elements of the semantic vector, while the max pooling layer highlights the significant elements and avoids overfitting [16].

**Global Semantic Feature Learning.** To learn the global semantic feature from the sliced code graph, we firstly use *Word2Vec* [19] as the embedding model to convert each token $v$ into an initial $d$-dimensional vector representation $x_v \in \mathbb{R}^d$. Then we use GGNN as the graph-level feature encoder to learn semantic features across the graph structure.

As GGNN is a recurrent neural network, the feature learning at time $t$ aggregate features for node $v$ from its neighbors along graph edges to get the aggregated features $a_v^{(t)}$:

$$a_v^{(t)} = A_v^T[h_1^{(t-1)T}, ..., h_m^{(t-1)T}] + b \tag{2}$$

where $b$ is a bias and $A_v$ is the adjacent matrix of node $v$ with learnable weights.

In addition, limited by the one-way message propagation mechanism of GNNs, rich contextual information that essential for vulnerability detection in smart contract may be ignored. Following existing works [1,4], we also propose to conduct the adjacency matrix for a graph with both incoming and outgoing edges for each node. The aggregated feature $a_v^{(t)}$ of node $v$ at time $t$ will be put into the gated recurrent unit (GRU) with the node's previous feature vector at time $t-1$ to get its current feature vector $h_v^{(t)}$:

$$h_v^{(t)} = GRU(a_v^{(t)}, h_v^{(t-1)}) \tag{3}$$

This computation iterate by $T$ times. Then, we use the state vectors computed by the last time step as the final node representations. The generated node features from the gated graph recurrent layers can be used as input to any prediction layer, e.g., for graph-level prediction in this work, and then the whole model can be trained in an end-to-end fashion. Considering that the importance of each node is different, we propose to use the hybrid attention graph pooling that combines self-attention pooling and global attention pooling to generate the final graph-level semantic feature $S_r$.

We first use a hierarchical self-attention graph pooling layer [14] to generate graph-level features from the most task-related nodes in a graph by self-attention mechanism. The self-attention pooling layer is consists of three blocks. In each block, it first calculates the attention score by a graph convention layer for each node. Then it picks up the top-k nodes based on the value of the attention score to construct a more task-related subgraph, which will be put into an average pooling to generate a subgraph-level feature. Finally, combining all these three blocks' subgraph-level features, it can get a graph-level feature $h_{top-K}$. Such hierarchical self-attention graph pooling can extract the most task-related features from all nodes, but just selecting top-$K$ nodes from the graph may lose some global information. Thus, we add a global attention pooling layer to complement the whole graph feature. In this layer, we propose to conduct the global graph feature by the weighted sum of all nodes features. As shown in Eq. (4), we use a Multi-Layer Perceptron (MLP) to compute the attention score for each node's feature

and use and a *Softmax* to get the weight for each node. By weighted sum all nodes' features, we get the global graph feature $h_{global}$ for each function.

$$h_{global} = \sum_{i=0}^{N} Softmax\left(MLP\left(h_i^{(T)}\right)\right) * h_i^{(T)} \tag{4}$$

Combining the most task-related graph feature and global graph feature, we get the final graph-level semantic feature $S_r$ by $S_r = h_{top-K} + h_{global}$.

**Feature Fusion.** After obtaining the local expert feature $P_r$ and the global semantic feature $S_r$ of the contract, we fuse them into the final feature $X_r$ by using a convolution layer and a max pooling layer:

$$X_r = P_r \oplus S_r \tag{5}$$

where $\oplus$ denotes concatenation operation.

### 3.3 Vulnerability Detection

With the learned integrated features $X_r$, the last step of SmartFuSE is to train a vulnerability detection model that will be applied to smart contracts under test. To this end, the learned integrated features are first fed into a network consisting of three fully connected layers and a sigmoid layer for the binary classification task:

$$\widetilde{y} = sigmoid(FC(X_r)) \tag{6}$$

where the fully connected layer $FC(\cdot)$ and the non-linear *sigmoid* layer produce the final estimated label $\widetilde{y}$. If the function is vulnerable, the value of $\widetilde{y}$ will be labeled as "1", otherwise labeled as "0".

## 4 Experiments

### 4.1 Research Questions

To evaluate the performance of SmartFuSE, we design the following research questions:

**RQ1:** To what extent smart contract vulnerability detection performance can SmartFuSE achieve?

**RQ2:** How do the integrated expert and semantic features affect the performance of smart contract vulnerability detection models?

**RQ3:** Can our proposed GGNN with hybrid attention pooling capture vulnerability-related program semantics?

### 4.2 Dataset

Following existing works [17,37], we built our evaluation benchmark by merging two real-world smart contract datasets, `ESC` and `VSC`.

### 4.3 Baselines

We selected five conventional analysis-based detectors and two state-of-the-art DL-based vulnerability detection approaches.

- **Oyente** [18] applies symbolic execution on the CFG of smart contract to check pre-defined vulnerable patterns.
- **Mythril** [20] combines multiple analysis techniques, such as concolic analysis and taint analysis, for smart contract vulnerability detection.
- **Smartcheck** [26] converts AST to a XML parse tree as an intermediate representation, and then check pre-defined vulnerability patterns on this intermediate representation.
- **Securify** [27] statically analyzes EVM bytecode to infer contract by Souffle Datalog solver to prove some pre-defined safety properties are satisfied.
- **Slither** [7] converts a contract to a specific intermediate representation named `SlithIR`, then doing the pre-defined vulnerability patterns match on this `SlithIR` representation.
- **Peculiar** [32] is a DL-based vulnerability detection tool which converts a contract to a crucial data flow graph and puts it into the GraphCodeBERT model for vulnerability detection.
- **TMP** [37] converts a contract to a normalized graph and then utilizes a deep learning model named temporal message propagation network for vulnerability detection.

### 4.4 Experimental Setup

**Implementation.** All the experiments are conducted on a server with an NVIDIA Tesla V100 GPU and an Intel(R) Core(TM) i9-12900k @3.90 GHz with 64 GB of RAM. The AST generate tool is implemented with typescript based on solc-typed-ast package, while CFG and PDG is conducted by Slither [7]. We use the pre-trained embedding model provided by SmartEmbed [8] to do the token embedding, and the GGNN model is implemented with python based on the dgl library [28]. The dimension of the vector representation of each node is set to 128 and the dropout is set to 0.2. ADAM [13] optimization algorithm is used to train the model with the learning rate of 0.001. The pool rate for self attention pooling layer is 0.5.

**Evaluation Metrics.** We apply the following four widely used evaluation metrics to measure the effectiveness of our approach and the other competitors.

- ***Accuracy*** **(Acc)** evaluates the performance that how many instances can be correctly labeled. It is calculated as: $Acc = \frac{TP+TN}{TP+FP+TN+FN}$.
- ***Precision*** **(Pre)** is the fraction of true vulnerabilities among the detected ones. It is defined as: $Pre = \frac{TP}{TP+FP}$.
- ***Recall*** **(Rec)** measures how many vulnerabilities can be correctly detected. It is calculated as: $Rec = \frac{TP}{TP+FN}$.
- ***F1-score*** **(F1)** is the harmonic mean of *Recall* and *Precision*, and can be calculated as: $F1 = 2 * \frac{Rec*Pre}{Rec+Pre}$.

# 5 Experimental Results

## 5.1 Experiments for Answering RQ1

Table 3 shows the overall results (the best performances are also highlighted in bold.) of each baseline and SmartFuSE on smart contract vulnerability detection in terms of the aforementioned evaluation metrics. Overall, SmartFuSE outperforms all of the five referred analysis-based detectors and two DL-based approaches.

**Table 3.** Evaluation results in percentage compared with state-of-the-art detectors.

| Method | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| Oyente | 57.3 | 41.1 | 42.8 | 41.9 |
| Mythril | 53.9 | 64.7 | 36.4 | 46.6 |
| Securify | 50.5 | 53.2 | 55.2 | 54.2 |
| Smartcheck | 37.8 | 59.4 | 43.5 | 50.2 |
| Slither | 61.9 | 63.1 | 58.4 | 50.7 |
| Peculiar | 82.7 | 55.2 | 41.6 | 47.4 |
| TMP | 85.0 | 83.9 | 66.5 | 74.2 |
| SmartFuSE | **91.4** | **88.6** | **94.3** | **91.4** |

We can find that the performance of bytecode-level approaches (Oyente, Mythril, and Securify) is poor. The reason is that most semantic and syntax features are lost during the compilation of bytecode. Compared to pattern-based detection tools (i.e., slither, smartcheck), SmartFuSE still performs better because these pre-defined patterns are too simple or fixed to cover all situations. By contrast, our feature fusion model can automatically learn expert and semantic features from the representation of source code to detect different types of vulnerabilities.

In addition, we can also find that SmartFuSE outperforms two DL-based detection approaches (Peculiar [32] and TMP [37]) in terms of all evaluation metrics. The reason is that although both Peculiar and TMP utilize the graph to represent the smart contract code like SmartFuSE, neither of them can comprehensively and precisely capture the vulnerability-related syntax and semantic features inside the code. In particular, Peculiar [32] only uses the data flow alone to represent the smart contract code, which may lose several critical features such as control flow between statements. Thus, such a one-sided code representation approach makes the DL model hard to capture all potential vulnerability patterns from such insufficient vulnerability-related features.

### 5.2 Experiments for Answering RQ2

**Table 4.** Comparing results on vulnerability detection with different features.

| Setting | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| Expert Features | 86.9 | 84.3 | 90.2 | 87.1 |
| Semantic Features | 83.2 | 81.5 | 88.6 | 84.9 |
| SmartFuSE | **91.4** | **88.6** | **94.3** | **91.4** |

We set three training scenarios (i.e., expert features, semantic features, and their Fusion) to train SmartFuSE for assessing the effectiveness of integrated features on boosting vulnerability detection. The experimental dataset is set the same as the experiment of RQ1 (i.e., 80%-10%-10% for training, validation, and testing). The comparison results are reported in Table 4 and the best performances are highlighted in bold for each approach on three different settings.

Obviously, we can observe that both expert features and semantic features have their own advantages in building an accurate prediction model, and expert features seem to have a better understanding of code characteristics than semantic features in the domain of smart contract vulnerability detection, revealing that incorporating security patterns is necessary and important to improve the performance. Furthermore, we can observe that combining semantic features with expert patterns indeed achieves better results compared to their pure semantic features counterparts. For example, SmartFuSE respectively gains a 9.86% accuracy and 7.66% F1 improvements over its variant with the pure neural network model, demonstrating the effectiveness of combining semantic features with expert patterns.

### 5.3 Experiments for Answering RQ3

**Table 5.** Comparing results on vulnerability detection with different pooling layers.

| Setting | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| Sum Pooling | 75.7 | 74.7 | 83.6 | 87.1 |
| Avg Pooling | 80.1 | 78.4 | 87.4 | 84.9 |
| Global Attention Pooling | 83.8 | 81.6 | 89.5 | 87.1 |
| Self Attention Pooling | 87.6 | 84.2 | 92.6 | 84.9 |
| SmartFuSE | **91.4** | **88.6** | **94.3** | **91.4** |

We further investigate the impact of our graph feature learning module with a hybrid attention pooling layer by comparing it with its variant. Towards this,

we use the pooling without attention, such as sum pooling and average pooling as the baseline, then conduct ablation experiments targeting the hybrid attention mechanism (i.e., respectively removing self-attention and global attention).

The comparison of our model with different pooling layers are shown in Table 5. We can see that our hybrid attention graph pooling layer get significant improvements in smart contract vulnerability detection task. The experimental results show that using pooling with attention mechanism can be more effective than other no-attention pooling layers. In addition, we can observe that compared to global and self attention pooling, the hybrid attention pooling is more effective than using them alone. The reason is that these two attention mechanisms are complementary, and combining them can improve the effectiveness.

## 6 Related Work

Traditional approaches employ static analysis or formal approaches to detect vulnerabilities [2,7,10,26]. For example, Slizer [7] and SmartCheck [26] are detectors of vulnerability patterns, which can perform static analysis on many types of source code. Bhargavan et al. [2] proposes a language-based formal approach to verify the safety and the functional correctness of smart contracts. Hirai et al. [10] proposed an interactive theorem prover to verify some safety properties of Ethereum smart contracts.

Another aspect of the work depends on symbol analysis and dynamic execution. Oyente [18] is the first symbol execution tool, which directly works with EVM byte code without access to corresponding source code. Zeus [12] employs both abstract interpretation and symbolic model checking to verify the correctness and fairness of smart contracts. ContractFuzzer [11] randomly generates test cases to identify vulnerabilities through fuzzing and runtime behavior monitoring during execution. Osiris [25] combined symbol execution and stain analysis to detect smart contract vulnerabilities related to integers.

Recently, several studies have demonstrated the effectiveness of Deep Learning (DL) in automated smart contract vulnerability detection. Qian et al. [24] proposed a novel attention-based BLSTM model to precisely detect reentrancy bugs. Zhuang et al. [37] combines contract graph and graph neural networks to detect three common smart contract vulnerabilities.

## 7 Conclusion

In this paper, we propose a novel approach SmartFuSE, which fully utilizes both expert features and semantic features of source code to build a performance-better model for smart contract vulnerability detection. We also explore the possibility of using graph neural networks with hybrid attention mechanism to learn precise graph features from code graphs, which contains rich vulnerability-specific program semantics. Extensive experimental results show that our proposed approach can significantly outperform existing detection approaches.

## References


1. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: Proceedings of the 6th International Conference on Learning Representations (ICLR) (2018)
2. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016, pp. 91–96. ACM (2016)
3. Cai, J., Li, B., Zhang, J., Sun, X., Chen, B.: Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. J. Syst. Softw. **195**, 111550 (2023)
4. Cao, S., Sun, X., Bo, L., Wei, Y., Li, B.: BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. **136**, 106576 (2021)
5. Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C.: MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), pp. 1456–1468. ACM (2022)
6. Falkon, S.: The story of the DAO - its history and consequences (2017)
7. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB@ICSE), pp. 8–15. IEEE / ACM (2019)
8. Gao, Z., Jiang, L., Xia, X., Lo, D., Grundy, J.: Checking smart contracts with structural code embedding. IEEE Trans. Software Eng. **47**(12), 2874–2891 (2021)
9. Hang, L., Kim, D.: Reliable task management based on a smart contract for runtime verification of sensing and actuating tasks in IoT environments. Sensors **20**(4), 1207 (2020)
10. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
11. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 259–269. ACM (2018)
12. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2018)
13. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: Proceedings of the 3rd International Conference on Learning Representations (ICLR) (2015)
14. Lee, J., Lee, I., Kang, J.: Self-attention graph pooling. In: Proceedings of the 36th International Conference on Machine Learning (ICML), vol. 97, pp. 3734–3743 (2019)
15. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: Proceedings of the 4th International Conference on Learning Representations (ICLR) (2016)
16. Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., Wang, X.: Combining graph neural networks with expert knowledge for smart contract vulnerability detection. arXiv preprint arXiv:2107.11598 (2021)

17. Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., Wang, X.: Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans. Knowl. Data Eng. **35**(2), 1296–1310 (2023)
18. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 254–269. ACM (2016)
19. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NeurIPS), pp. 3111–3119 (2013)
20. Mueller, B.: A framework for bug hunting on the ethereum blockchain (2017)
21. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
22. Park, J., Youn, T., Kim, H., Rhee, K., Shin, S.: Smart contract-based review system for an IoT data marketplace. Sensors **18**(10), 3577 (2018)
23. Pierro, G.A., Tonelli, R., Marchesi, M.: An organized repository of ethereum smart contracts' source codes and metrics. Future Internet **12**(11), 197 (2020)
24. Qian, P., Liu, Z., He, Q., Zimmermann, R., Wang, X.: Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access **8**, 19685–19695 (2020)
25. Tann, W.J., Han, X.J., Gupta, S.S., Ong, Y.: Towards safer smart contracts: a sequence learning approach to detecting vulnerabilities. arXiv preprint arXiv:1811.06632 (2018)
26. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: static analysis of ethereum smart contracts. In: Proceedings of the 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB@ICSE), pp. 9–16. ACM (2018)
27. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 67–82. ACM (2018)
28. Wang, M., et al.: Deep graph library: a graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)
29. Wei, Y., Sun, X., Bo, L., Cao, S., Xia, X., Li, B.: A comprehensive study on security bug characteristics. J. Softw. Evol. Process. **33**(10), e2376 (2021)
30. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
31. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014)
32. Wu, H., et al.: Peculiar: smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 378–389. IEEE (2021)
33. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2021)
34. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: Proceedings of the 35th IEEE Symposium on Security and Privacy (SP), pp. 590–604. IEEE Computer Society (2014)
35. Zhang, Y., Kasahara, S., Shen, Y., Jiang, X., Wan, J.: Smart contract-based access control for the internet of things. IEEE Internet Things J. **6**(2), 1594–1605 (2019)

36. Zhou, Y., Liu, S., Siow, J.K., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS), pp. 10197–10207 (2019)
37. Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q.: Smart contract vulnerability detection using graph neural network. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI), pp. 3283–3290 (2020)