Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion

Zhenguang Liu1*, Peng Qian1*, Xiang Wang2, Lei Zhu3, Qinming He1 and Shouling Ji1†
1Zhejiang University
2National University of Singapore
3Shandong Normal University
{liuzhenguang2008, messi.qp711, xiangwang1223, leizhu0608}@gmail.com, {hqm, sji}@zju.edu.cn

Abstract

Smart contracts hold digital coins worth billions of dollars, their security issues have drawn extensive attention in the past years. Towards smart contract vulnerability detection, conventional methods heavily rely on fixed expert rules, leading to low accuracy and poor scalability. Recent deep learning approaches alleviate this issue but fail to encode useful expert knowledge. In this paper, we explore combining deep learning with expert patterns in an explainable fashion. Specifically, we develop automatic tools to extract expert patterns from the source code. We then cast the code into a semantic graph to extract deep graph features. Thereafter, the global graph feature and local expert patterns are fused to cooperate and approach the final prediction, while yielding their interpretable weights. Experiments are conducted on all available smart contracts with source code in two platforms, Ethereum and VNT Chain. Empirically, our system significantly outperforms state-of-the-art methods. Our code is released.

1 Introduction

Blockchain has attracted extensive attention in the past few years. The worldwide miners (bookkeeping nodes) obey a consensus protocol to maintain a secure and shared transaction ledger, which is termed a blockchain [Hewa et al., 2020]. In the blockchain network, the consensus protocol enforces the transactions immutable once recorded in the distributedly copied ledger, endowing the blockchain with tamper-resistant and decentralization nature.

Smart contracts are programs running on top of a blockchain system [Wang et al., 2019; Luu et al., 2016]. A smart contract can be specially designed by developers to implement arbitrary rules for managing digital assets. Attributing to the immutable nature of blockchain, a smart contract cannot be updated once deployed. Thus, the defined rules of a smart contract are formulated as program code and are automatically executed, which is impartial for all parties that interact with the contract. Smart contracts make the automatic execution of contract terms possible, facilitating complex decentralized applications (DApps) [Ding et al., 2019].

So far, millions of smart contracts have been deployed on various blockchain platforms, controlling digital currency worth more than 10 billion dollars. Holding so much wealth, however, makes smart contracts attractive enough to malicious attackers. In 2016, attackers exploited the reentrancy vulnerability of The DAO contract1 to steal Ether (i.e., Cryptocurrency of Ethereum) worth 60 million dollars. This case is not isolated and several security vulnerabilities of smart contracts are disclosed every year [Lorenz et al., 2018].

The main issues that may easily lead to smart contract vulnerabilities are twofold. First, the programming languages and tools are still new and crude, which leaves plenty of rooms for misunderstandings in the built-in functions and tools [Luu et al., 2016]. Second, due to the immutable nature of smart contracts, developers are required to anticipate all possible states (e.g., stack status) and environments that the code may encounter in the future, which is obviously difficult.

Existing methods on smart contract vulnerability detection can be roughly cast into two categories. The first line of work [Luu et al., 2016; Tsankov et al., 2018; Jiang et al., 2018] utilized classical static analysis and dynamic execution techniques to identify vulnerabilities. Unfortunately, they 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. Meanwhile, crafty attackers may easily bypass the fixed patterns using small tricks. Another line of work [Tann et al., 2018; Zhuang et al., 2020] explored using deep learning models to deal with complex contract data, having achieved much improved accuracy. Due to the black-box nature, however, they fail to encode useful expert knowledge and mostly have poor interpretability. This motivates us to consider whether we could combine neural networks with classical expert patterns, where neural networks contribute their ability to handle the complex code semantic graph while expert patterns contribute precise and valuable local information. More importantly, we seek an explainable solution which could tell the

1The DAO contract, 2016. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413

*The first two authors have equal contribution.
†Corresponding author.
function withdraw(uint sum) {
  if (Balance[msg.sender] < sum) {
    throw;
  }
  require(msg.sender.call.value(sum)());
  Balance[msg.sender] -= sum;
}

withdraw(uint sum) {
  mark datasets, demonstrating significant improvements over
ments are conducted on all the 40k contracts in two bench-

tive multi-encoder network to interpretably fuse the
function code. (3) Finally, we propose an atten-
transfer the deep graph feature. (3) Finally, we propose an atten-
control-flow and data-flow semantics of the function code.
Upon the graph, a graph neural network is employed to ex-
depth the deep graph feature. (3) Finally, we propose an attentive
multi-encoder network to interpretably fuse the global
graph feature and local expert patterns. Extensive experi-
ments are conducted on all the 40k contracts in two benchmark
datasets, demonstrating significant improvements over
state-of-the-art: accuracy from 84% to 90%, 83% to 87%,
75% to 80% on three types of vulnerabilities respectively.
More importantly, our model is able to explain its label pre-
diction, give warnings of high weighted local patterns, and
provide a grand picture of the significance of different fea-
tures.

Contributions. The key contributions of this work are: 1) We
investigate combining vulnerability-specific expert pat-
terns with neural networks in an explainable way. To the best
of our knowledge, we are the first to prob the combination in-
terpretable. 2) In the method, we present a simple but effective
multi-encoder network for feature fusion. 3) Our method
sets the new state-of-the-art and provides novel insights. To
facilitate future research, our implementations are released at
https://github.com/Messi-Q/AMEVulDetector. We would
like to point out that different from [Liu et al., 2021], this
work focuses mainly on the explainability of the expert pat-
tern and deep graph feature combination, and offers a grand
picture on the importance of different features.

2 Problem

Given the source code of a smart contract, we seek to de-
vlop a fully automatic approach that can detect vulnerability
at fine-grained function level and is interpretable. In other
words, presented with a smart contract function \( f \), we are to
predict its label \( l \) and output the associated weights \( \{w_i\}_{i=1}^n \)
simultaneously, where \( l = 1 \) represents \( f \) has a vulnerability
of a certain type and \( l = 0 \) denotes \( f \) is safe. Weight \( w_i \) ex-
plains the importance of the \( i^{th} \) feature in predicting the label.
In this work, we concentrate on three common vulnerabilities:

(1) Reentrancy is a well-known vulnerability that has brought
about the notorious DAO attack. When a smart
contract function \( f_1 \) transfers money to a recipient contract
\( C \), the fallback function \( f_2 \) of \( C \) will be automatically trig-
gered. Function \( f_2 \) may invoke \( f_1 \) for conducting an in-
valid second-time money transfer. Since the current
execution of \( f_1 \) waits for the first-time transfer to finish, the
balance of \( C \) is not reduced yet, \( f_1 \) thus may wrongly be-
lieve that \( C \) still has enough balance and transfers money
to \( C \) again. More specifically, the expected execution trace
is \( f_1 \rightarrow C \rightarrow f_2 \rightarrow \text{end} \), whereas the actual
trace is \( f_1 \rightarrow C \rightarrow f_2 \rightarrow f_1 \rightarrow C \rightarrow f_2 \rightarrow \text{end} \). Exploiting the reentrancy vulner-
ability, an attacker may succeed in obtaining 20 Ether al-
though his/her balance is 10 Ether. Note that attackers can
steal more money by invoking \( f_1 \) more than one time, which
can be exemplified as \( f_1 \rightarrow C \rightarrow f_2 \rightarrow f_1 \rightarrow C \rightarrow f_2 \rightarrow \text{end} \).

(2) Block timestamp dependence happens when a function
uses block timestamp as a condition to perform critical op-
erations, e.g., using block.timestamp of a future block as the
source to generate random numbers so as to determine the
winner of a game. The miner who mines the block has the
freedom to set the timestamp of the block as long as it is
within a short time interval [Jiang et al., 2018]. Thus, miners
may manipulate block timestamp to gain illegal benefits.

(3) Infinite loop is conventionally considered as a loop bug
which unintentionally iterates forever, e.g., a for loop with no
exit condition. Distinct from conventional programs, users
have to pay a fee for executing each line of smart contract
code. The fee is approximately proportional to how much
code needs to run. For a function with infinite loop, its ex-
ecution will run out of gas and be aborted. In such a case, the
execution consumes a lot of gas but all the gas is consumed
in vain since the execution is unable to change any state.

3 Method

The overview of the proposed system is illustrated in Fig. 1,
which consists of three components: 1) a local expert pattern
tool, which extracts expert patterns of a specific vulner-
bility from the function code; 2) a graph construc-
tion and normalization module, which transforms the func-
tion code into a code semantic graph; and 3) an attentive

Smart contract function (a) Local pattern extraction (b) Global graph construction and normalization (c) Attentive multi-encoder network

Figure 1: The overall architecture of our proposed method. (a) The local expert pattern extraction tool for extracting vulnerability-specific expert patterns. (b) The graph construction and normalization module for transforming the code into a global semantic graph. (c) The attentive multi-encoder network, which combines expert patterns and the graph feature for vulnerability detection and outputting explainable weights.
multi-encoder network that combines local expert patterns and the global graph feature for vulnerability detection and outputs explainable weights. In what follows, we introduce the three components one by one.

### 3.1 Local Expert Pattern Extraction

Following [Liu et al., 2021], we design corresponding expert patterns for three types of vulnerabilities respectively. Then, we implement a fully automatic tool to extract expert patterns from the function code. Specifically, the patterns for different vulnerabilities are defined as:

**Reentrancy.** Technically, the reentrancy vulnerability occurs when a call.value invocation (i.e., a built-in money transfer function) can call back to itself through a chain of calls. That is, call.value is successfully re-entered to perform unexpected repeat money transfers. For the reentrancy vulnerability, we design three local patterns. (1) *enoughBalance* concerns whether there is a check on the sufficiency of the user balance before transferring to a user. (2) *callValueInvocation* models whether there exists an invocation to call.value in the function. (3) *balanceDeduction* checks whether the user balance is deducted after money transfer, which considers the fact that the money stealing can be avoided if the user balance is deducted each time before money transfer.

**Block timestamp dependence.** Generally, the timestamp dependence vulnerability exists when a smart contract is conventionally considered as using block.timestamp as part of the conditions to perform critical operations [Jiang et al., 2018]. We design three local patterns for the timestamp dependence vulnerability. (1) *timestampInvocation* models whether there exists an invocation to opcode block.timestamp in the function. (2) *timestampAssign* checks whether the value of block.timestamp is assigned to other variables or passed to a function as a parameter, namely whether block.timestamp is actually used. (3) *timestampContaminate* verifies if block.timestamp may contaminate the triggering condition of a critical operation (e.g., money transfer).

**Infinite loop.** Specifically, we define three local patterns for the infinite loop vulnerability as follows. (1) *loopStatement* checks whether the function exists a loop statement such as for and while. (2) *loopCondition* validates whether the exit condition can be reached. For example, for a while loop, its exit condition $i < 9$ cannot be reached if $i$ is never updated in the loop. (3) *selfInvocation* concerns whether the function invokes itself and the self-invocation is not in an if statement. This considers the fact that if the self-invocation statement is not in an if statement, the self-invocation loop will never terminate.

**Implementations.** Our open-sourced tool to extract the designed local expert patterns is released on Github. These patterns are consistent with those in our previous work [Liu et al., 2021]. As introduced in [Liu et al., 2021], the pattern extraction tool scans the source code of a function several times. Particularly, simple patterns such as callValueInvocation and timestampInvocation can be directly extracted by keyword matching. Patterns such as enoughBalance, balanceDeduction, loopStatement, timestampAssign, loopCondition, and selfInvocation are obtained by syntactic and semantic analysis. Complex pattern timestampContaminate is extracted by taint analysis, where we follow the traces of the data flow and flag all the variables that may be affected along the traces.

### 3.2 Graph Construction and Normalization

One may directly use combinations of aforementioned patterns to predict whether the function has a certain vulnerability. However, these fixed patterns are shown to have difficulties in handling relatively complex attacks and are trivial to be bypassed by adversaries [Qian et al., 2020]. Therefore, we further propose to model the control flow and data flow of the entire source code into a semantic graph, and adopt graph neural networks to handle it. Thereafter, the extracted global graph feature and the local expert patterns could supplement each other towards a more precise and explainable label prediction.

**Graph construction.** Different program elements in a function are not of equal importance in detecting vulnerabilities. Therefore, we extract two categories of nodes, i.e., core nodes and normal nodes. (1) Core nodes symbolize key invocations and variables in the function code, which are critical for detecting a specific vulnerability. For instance, for reentrancy vulnerability, (i) an invocation to call.value, (ii) an invocation to a money transfer function that contains a call.value invocation, and (iii) the variable that corresponds...
to user balance, are treated as core nodes. We summarize the core nodes for detecting the three vulnerabilities in Table 1. (2) Invocations and variables that are not extracted as core nodes are modeled as normal nodes, which play an auxiliary role in detecting vulnerabilities. We also construct an extra fallback node to stimulate the fallback function of a virtual attack contract. The fallback node can interact with the function under test and is considered as a normal node. It is worth mentioning that distinct from prior works such as [Zhuang et al., 2020], which merely model key invocations as core nodes, we propose to further extract key variables as core nodes, given that they are undoubtedly important in detecting vulnerabilities. To characterize rich connections between different nodes, we construct three categories of semantic edges, namely control flow, data flow, and fallback edges. Each edge describes a path that might be traversed through by the function under test, and the temporal number of the edge is set according to its sequential order in the function code. We summarized key semantic edges in Table 2.

Graph normalization. It is worth mentioning that different functions corresponding to distinct code semantic graphs, bringing difficulties in training a graph neural network. Moreover, current graph neural networks are inherently flat when propagating information, ignoring that different nodes are not of equal importance. Therefore, we propose to normalize the graph following that of [Zhuang et al., 2020] to remove all normal nodes and merge their features to the nearest core nodes. A simplified example for graph construction and normalization is given in Fig. 1(b).

3.3 Attentive Multi-Encoder Network

For the extracted local expert patterns, we adopt the multiple MLPs (multilayer perceptrons) to encode them into feature vectors. For the normalized global code semantic graph, we utilize a temporal-message-propagation (TMP) graph neural network to transform it into a deep graph feature. Then, the expert pattern features and the graph feature are fused using an attentive multi-encoder network to give the overall label prediction \( \hat{y} \), which is illustrated in Fig. 2.

Local pattern feature extraction. Each expert pattern formulates an elementary factor closely related to a specific vulnerability. We utilize a one-hot vector to represent each pattern, and append a digit 0/1 to indicate whether the function under test has this pattern. The vectors for all patterns related to a specific vulnerability are fed into multiple MLPs, each takes care of one pattern. The outputs of the MLPs are concatenated and passed into a feed-forward network to compute the label prediction of the function. This network is trained so that each MLP learns to extract the feature \( p \in \mathbb{R}^d \) of a pattern.

Global graph feature extraction. To extract the graph feature from the normalized code semantic graph, we use a graph neural network that consists of a message propagation phase and an aggregation phase [Zhuang et al., 2020]. In the message propagation phase, the network passes information along the edges successively by following their sequential orders in the code, while the aggregation phase outputs the global graph feature \( g \in \mathbb{R}^d \) by aggregating the final states and original states of all nodes. Following [Zhuang et al., 2020], at each time step \( j \), message flows through the \( j \)-th edge \( e_j \) and updates the hidden state of the end node of \( e_j \) by absorbing information from edge \( e_j \) and the start node of \( e_j \). After successively traversing all edges, we extract the graph feature by aggregating the hidden states of all nodes. The original hidden state \( h_i^0 \) and the final hidden state \( h_i \) of each node are informative in the vulnerability detection task. Therefore, we can obtain the final global graph feature \( g \) by:

\[
\begin{align}
    h_i &= h_i^0 \oplus h_i^T \\
    Gate_i &= \sigma(M_1(relu(b_1 + M_2h_i))) + b_2 \\
    Output_i &= \sigma(M_3(relu(b_3 + M_4h_i)) + b_4) \\
    g &= FC\left( \sum_{i=1}^{N} (Gate_i \odot Output_i) \right)
\end{align}
\]

where \( \oplus \) and \( \odot \) denote concatenation and element-wise product. Matrix \( M_j \) and bias vector \( b_j \) are network parameters, \( N \) is the number of nodes, \( \sigma \) is the softmax activation layer, and \( FC \) is a fully connected layer. Motivated readers may also refer to [Zhuang et al., 2020] for more details of the message propagation and aggregation phases.

Fusion network. As shown in Fig. 2, the extracted local pattern features \( \{p_i\}_{i=1}^{K} \) and the global graph feature \( g \) go through a self-attention mechanism and a cross-attention mechanism to output the label prediction \( \hat{y} \) and the associated weights \( \{w_i\}_{i=1}^{K} \) for all features. Since the local expert pattern features and the global graph feature are heterogeneous,
the self-attention mechanism learns to preliminarily balance different features by computing the coefficients for all features. The cross-attention mechanism then fuses different features to predict the label.

**Self attention mechanism.** Formally, the processes of the self attention for $p_i$ and $g$ are given by:

$$c_g = FC(g), \quad g = c_g \odot g \quad (5)$$

$$c_{p_i} = FC(p_i), \quad p'_i = c_{p_i} \odot p_i \quad (6)$$

where $c_g$ and $c_{p_i}$ are coefficient vectors, $\odot$ denotes the element-wise product, $g'$ and $p'_i$ denote the updated features.

**Cross attention mechanism.** After the self attention, we obtain new features $g$ and $\{p'_i\}_{i=1}^k$. Next, we consider combining them to detect vulnerabilities. As shown in Fig. 2, each feature goes through an encoder, mapping $g$ and $\{p'_i\}_{i=1}^k$ into $g'$ and $\{p_i\}_{i=1}^k$, respectively. Then, all the mapped features are concatenated and fed into an MLP, generating a final semantic vector $v$. The label $\hat{y}$ is then conveniently computed by $\hat{y} = round(sigmoid(v))$. Technically, we believe that the attention weights could highlight the feature importance. The inner product between each feature and $v$, therefore, is employed to compute the interpretable weights for the global graph feature $g$ and each local expert pattern feature $p_i$.

$$w_g = \frac{exp(g \cdot v)}{exp(g \cdot v) + \sum_{i=1}^{k} exp(p'_i \cdot v)} \quad (7)$$

$$w_{p_i} = \frac{exp(p'_i \cdot v)}{exp(g \cdot v) + \sum_{i=1}^{k} exp(p'_i \cdot v)} \quad (8)$$

where $w_g$ denotes the weight of $g$, $w_{p_i}$ is the weight of $p_i$, and $\cdot$ represents inner product.

### 4 Experiments

In this section, we empirically evaluate our proposed method on two benchmark datasets, namely Ethereum smart contract dataset (ESC) and VNT Chain smart contract dataset (VSC). We seek to answer the following research questions:

- **RQ1:** Can the proposed method effectively detect the three types of vulnerabilities? How is its performance against state-of-the-art tools and neural network-based methods?
- **RQ2:** Is the method able to provide interpretability in vulnerability detection? Can we obtain new insights from it?
- **RQ3:** How do the different components affect the performance of the proposed approach?

Next, we first present the experimental settings, followed by answering the above research questions one by one.

#### 4.1 Experimental Settings

**Datasets.** (1) ESC dataset contains 307,396 functions from 40,932 Ethereum smart contracts. In the dataset, 5,013 functions contain at least one invocation to `call.value`, making them potentially affected by reentrancy vulnerability. 4,833 functions have the `block.timestamp` statement that may cause timestamp dependence vulnerability. (2) VSC dataset consists of 13,761 functions from 4,170 VNT Chain smart contracts. Around 2,925 functions have loop statements.

**Implementation details.** All experiments are conducted on a computer equipped with an Intel Core i7 CPU at 3.7GHz, a GPU at 1080Ti, and 32GB Memory. The expert pattern extraction tool and the graph construction tool are implemented with Python, while the neural networks are implemented with TensorFlow. Following prior works, we randomly select 80% of the functions as the training set and the other 20% as the test set for each dataset. The encoders in Fig. 2 are implemented using three fully connected layers. The hidden state sizes of the self attention and encoder layers are 200 and 100, respectively.

#### 4.2 Performance Comparison (RQ1)

In this section, we compare our proposed approach against state-of-the-art tools and available neural network-based methods. Following existing work [Zhuang et al., 2020], we conduct experiments for the reentrancy and timestamp dependence vulnerability on ESC, and evaluate the infinite loop vulnerability on VSC. Metrics accuracy, recall, precision, and F1-score are all engaged in the comparisons.

**Comparison with Conventional Detection Tools**

We first compare our method AME with existing smart contract vulnerability detection tools including Smartcheck [Tikhomirov et al., 2018], Oyente [Luu et al., 2016], Mytril [Mueller, 2017], Securify [Tsankov et al., 2018], and Slithier [Feist et al., 2019]. Quantitative results are summarized in Table 3.

As shown in the left of Table 3, we observe that: 1) conventional tools have not yet achieved a satisfactory accuracy on the reentrancy vulnerability detection. In particular, state-of-the-art tools Securify and Slithier only achieve 71.89% and 77.12% accuracies. 2) Our method significantly outperforms the existing tools in reentrancy vulnerability detection. More specifically, AME achieves a 90.19% accuracy, gaining a 13.07% accuracy improvement over the state-of-the-art tool. Empirical evidences clearly reveal the effectiveness of our method.

Next, we evaluate all the methods on timestamp dependence vulnerability. The comparison results are shown in the middle of Table 3. State-of-the-art tool Slithier obtains a 74.20% accuracy, which is quite low. This may stem from the fact that most conventional tools handle the timestamp dependence vulnerability by blindly checking whether there is a `block.timestamp` statement in the function, while ignoring whether the timestamp can truly affect a critical operation. Further, it is worth pointing out that AME keeps delivering the best performance in terms of all the four metrics. Significantly, AME gains a 12.32% accuracy improvement over the state-of-the-art tool.

We further evaluate our method on the infinite loop vulnerability. Specifically, we compare our methods against existing infinite loop detection methods including Jolt [Carbin et al., 2011], SMT [Kling et al., 2012], PDA [Ibing and Mai, 2015], and Looper [Burnim et al., 2009]. Quantitative results are illustrated in the right of Table 3. We observe that AME consistently outperforms other methods by a large margin.

By looking into the implementations of classical tools, we found that: 1) they heavily rely on a few fixed expert rules to
Table 3: Performance comparison. A total of sixteen methods are investigated in the comparisons. ‘–’ denotes not applicable.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Reentrancy</th>
<th>Timestamp dependence</th>
<th>Infinite Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acc(%)</td>
<td>Recall(%)</td>
<td>Precision(%)</td>
</tr>
<tr>
<td>Smartcheck</td>
<td>52.97</td>
<td>32.08</td>
<td>25.00</td>
</tr>
<tr>
<td>Oyente</td>
<td>61.62</td>
<td>54.71</td>
<td>38.16</td>
</tr>
<tr>
<td>Mythril</td>
<td>60.54</td>
<td>71.69</td>
<td>39.58</td>
</tr>
<tr>
<td>Security</td>
<td>71.89</td>
<td>56.60</td>
<td>50.85</td>
</tr>
<tr>
<td>Slither</td>
<td>77.12</td>
<td>74.28</td>
<td>68.42</td>
</tr>
<tr>
<td>Vanilla-RNN</td>
<td>49.64</td>
<td>58.78</td>
<td>49.82</td>
</tr>
<tr>
<td>LSTM</td>
<td>53.68</td>
<td>67.82</td>
<td>51.65</td>
</tr>
<tr>
<td>GRU</td>
<td>54.54</td>
<td>71.30</td>
<td>53.10</td>
</tr>
<tr>
<td>GCN</td>
<td>77.85</td>
<td>78.79</td>
<td>70.02</td>
</tr>
<tr>
<td>DR-GCN</td>
<td>81.47</td>
<td>80.89</td>
<td>72.36</td>
</tr>
<tr>
<td>TMP</td>
<td>84.48</td>
<td>82.63</td>
<td>74.06</td>
</tr>
<tr>
<td>AME</td>
<td>90.19</td>
<td>89.69</td>
<td>86.25</td>
</tr>
</tbody>
</table>

Figure 3: Weight statistics of different features. Left: ratio of having a weight higher than $\sigma$. Right: ratio of having the maximum weight.

Figure 4: Mean and deviation of the weights of different features.

4.3 Interpretability Evaluation (RQ2)

We now study the interpretability of the proposed AME network, and present novel insights on weight distributions of different features in vulnerability detection. We also present a case study to facilitate understanding.

As presented in subsection 3.1, we defined three local patterns for each vulnerability. We denote the three vulnerability-specific patterns as Loc-pattern1, Loc-pattern2, and Loc-pattern3, respectively. For example, Loc-pattern1 is enoughBalance for reentrancy vulnerability and is timestampInvocation for timestamp dependence vulnerability. The three patterns and the global graph feature are used to predict whether the function has the specific vulnerability.

Interestingly, to figure out which features contribute the most to the detection of a specific vulnerability, we record the number that a feature possesses a weight (obtained by Eq. 7 or Eq. 8) greater than a preset threshold $\sigma = 0.25$ over all tested
functions. The statistics are visualized in the left of Fig. 3. We observe that the global graph feature usually possesses a high weight, revealing its leading role in the label prediction. Moreover, in reentrancy vulnerability detection, Loc-pattern2 (callValueInvocation) and Loc-pattern3 (balanceDeduction) rank second and third in the possibility of having a high weight, while Loc-pattern2 (timestampAssign) and Loc-pattern3 (timestampContaminate) often have high weights in timestamp dependence detection. It is worth pointing out that once a function is predicted to have a specific vulnerability by our method, besides telling the user how significant (weight) each feature contributes to the prediction, we can also warn him/her of the expert patterns that get high weights to help find bugs. The statistics over a large number of functions also allow developers build a grand picture of the whole system.

We also compute the number that a feature has the maximum weight among all features over all tested functions. The visualized results are demonstrated in the right of Fig. 3. From the histogram, we notice that the global graph feature obtains the maximum weight most times for the three vulnerabilities. For each feature, we also illustrate the mean and standard deviation of its weight in Fig. 4. We see that each local expert pattern possesses a considerable weight, but the average attention weight of the global graph feature is the highest. This may come from the fact that the global graph conveys the control- and data-dependencies of a specific vulnerability and contains richer global semantic information than local expert patterns.

**Case study.** We further present a case study in Fig. 5, where the `withdraw` function is a real-world smart contract function that has a reentrancy vulnerability. We analyzed the code to detect whether it has the three vulnerabilities. The left of Fig. 5 illustrates the global graph and the three local patterns of reentrancy. The right of Fig. 5 shows the interpretable weights of the graph feature and each local pattern feature. For example, the method detects the function has the reentrancy vulnerability. To make the detection decision, the weight of the global graph feature is 0.27, the weights of Loc-pattern1 (enoughBalance), Loc-pattern2 (callValueInvocation), and Loc-pattern3 (balanceDeduction) are 0.23, 0.26, and 0.24, respectively. Thus, our system is able to clearly explain the reasons behind the predictions.

### 4.4 Effects of Removing Expert Patterns and Graph Feature (RQ3)

By default, we combine the graph feature with expert patterns for vulnerability detection. We are interested in exploring the effect of removing them respectively. To this aim, we first remove the code semantic graph construction module, and only use the expert pattern features as the inputs. We denote this variant as AME-RG (RG represents removing graph feature). We also try removing the expert pattern extraction module and using the graph feature only, which is denoted as AME-RP (RP represents removing patterns, which is almost identical to the TMP). Fig. 6 demonstrates the comparison results on three vulnerabilities, where the solid curves demonstrate the accuracy of AME over different epochs, and the dashed curves show its variants. Different colors represent different vulnerabilities. Clearly, the performance of AME is consistently better compared to its variants across all epochs, revealing that combining local expert patterns with global graph features is necessary and important to improve the detection performance. Moreover, removing the neural network extracted graph feature exhibits a higher performance drop than removing the expert patterns.
5 Related Work

Early work on smart contract vulnerability detection verifies smart contracts by employing formal methods [Bhargavan et al., 2016; Hirai, 2017; Grishchenko et al., 2018]. For example, [Bhargavan et al., 2016] introduces a framework, translating Solidity code and the EVM (Ethereum Virtual Machine) bytecode into the input of an existing verification system. [Hirai, 2017] proposes a formal model for EVM (Ethereum Virtual Machine) and reasons smart contracts using the Isabelle/HOL tool. Another stream of work relies on symbolic analysis and dynamic execution. Oyente performs symbolic execution on contract functions and flags bugs based on simple patterns. Zeus [Kalra et al., 2018] leverages abstract interpretation and symbolic model checking. [Tsankov et al., 2018] introduces compliance (negative) and violation (positive) patterns to filter false warnings. [Jiang et al., 2018] presents ContractFuzzer to identify vulnerabilities by fuzzing and runtime behavior monitoring during execution. Sereum [Rodler et al., 2019] uses taint analysis to monitor runtime data flows during smart contract execution for vulnerability detection. Recently, a few attempts have been made to study using deep neural networks. [Qian et al., 2020] constructs the sequential contract snippet and feeds them into the BLSTM-ATT model. [Zhuang et al., 2020] proposes to convert the source code into the contract graph and constructs graph neural networks as the detection model. [Wang et al., 2020] proposes extracting bigram features from operation codes of smart contracts. [Liu et al., 2021] proposes to combine expert rules with neural networks for improving the detection accuracy. However, [Liu et al., 2021] suffers from poor explainability and fail to investigate the significance of different features.

6 Conclusion

In this paper, we explore combining deep learning with classical expert patterns in an explainable way for smart contract vulnerability detection. This system consists of both neural networks and automatic expert pattern extraction tools. Interestingly, the model is able to obtain explainable fine-grained details and a grand picture of the weight distributions. Extensive experiments show that our method significantly outperforms state-of-the-art approaches. We believe our work is an important step towards explainable and accurate contract vulnerability detection.

Acknowledgements

This paper is supported by the Natural Science Foundation of Zhejiang Province, China (No. LQ19F020001), the National Natural Science Foundation of China (No. 61902348), and the Key R&D Program of Zhejiang Province (No. 2021C01104).

References


contract vulnerability detection. Transactions on Knowledge and Data Engineering, page Accepted, 2021.


