IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 9 June 2023, accepted 18 July 2023, date of publication 24 July 2023, date of current version 1 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3298048

RESEARCH ARTICLE

A Smart Contract Vulnerability Detection Mechanism Based on Deep Learning and Expert Rules

ZHENPENG LIU[1,2], MINGXIAO JIANG[2], SHENGCONG ZHANG[2], JIALIANG ZHANG[2], AND YI LIU[1]

[1]Information Technology Center, Hebei University, Baoding 071002, China
[2]School of Cyber Security and Computer, Hebei University, Baoding 071002, China

Corresponding author: Yi Liu (liuyi@hbu.edu.cn)

This work was supported in part by the National Natural Science Foundation of Hebei Province, China, under Grant F2019201427; and in part by the Ministry of Education of China's Fund for Integration of Cloud Computing and Big Data, Innovation of Science and Education (FII), under Grant 2017A20004.

ABSTRACT Traditional techniques for smart contract vulnerability detection rely on fixed expert criteria to discover vulnerabilities, which are less generalizable, scalable, and accurate. Deep learning algorithms help to address these issues, but most fail to encode true expert knowledge and remain interpretable. In this paper, we present a smart contract vulnerability detection mechanism that operates in phases with graph neural networks and expert patterns in deep learning to mutually address the deficiencies of the two detection approaches and improve smart contract vulnerability detection capabilities. Experiments show that our vulnerability detection mechanism outperforms the original deep learning model by an average of 6 points in detecting vulnerabilities and that the second stage of the checking mechanism can also block contract transactions containing dangerous actions at the Ethernet Virtual Machine (EVM) level and generate error reports for submission. This strategy helps to construct more stable smart contracts and to create a secure environment for smart contracts.

INDEX TERMS Smart contract, vulnerability detection, deep learning, EVM, expert knowledge.

I. INTRODUCTION

The concept of blockchain and digital currency has received a great deal of attention in recent years, owing to the gradual maturation of blockchain technology [1] and the rapid development of cryptocurrencies, where the synergy of decentralized consensus protocols [2] and proof-of-work mechanisms [2] ensures decentralized operations, transparency of transaction processes, and tamper-evident properties. These new technologies have resulted in the development of smart contracts, which are computer protocols meant to be disseminated, verified, or performed in an informational manner, serving as the foundation for a wide range of applications and services. Yet, the complexity of smart contracts is expanding due to the rapid growth of smart contracts, and the security challenges emerging from the frequent occurrence of smart contract vulnerabilities are becoming increasingly critical.

Smart contracts, as decentralized applications operating on the blockchain, are created with several blockchain-specific mechanisms [3], such as the gas mechanism, delegate call mechanism, exception passing mechanism, and some other particular smart contract mechanisms. Although these qualities have aided in the rapid spread and development of smart contracts on the blockchain, the availability of these particular processes has also resulted in different vulnerabilities in many smart contracts proposed based on them. In the case of Ethereum [4], although modern contracts can be updated by Create2 to prevent vulnerabilities, the maintenance cost of frequent contract updates is not easy and the damage caused by a vulnerability attack can no longer be recovered. It necessitates a thorough security examination of the

The associate editor coordinating the review of this manuscript and approving it for publication was Thanh Ngoc Dinh.

smart contracts to be deployed, with the goal of preventing problematic contracts from being distributed.

Existing smart contract vulnerability detection methods are broadly classified into two types: traditional detection methods that rely on expert rules formed by empirical knowledge and supplemented by some automated vulnerability detection tools, and deep learning-based vulnerability detection methods that have emerged in recent years. Conventional approaches rely on predefined expert guidelines, which are prone to error and are becoming increasingly difficult to supply the expanding demand. Furthermore, because of the fixed model, it is difficult to guarantee the vulnerability detection results for smart contracts written and generated with ulterior goals to purposefully bypass certain established restrictions, such as Oyente [5], Securify [6], and others. Deep learning-based models detect vulnerabilities with higher detection efficiency and better generalization, but they are highly dependent on datasets, can't easily encode valid expert knowledge due to the nature of black boxes, and most methods have inadequate interpretability [7].

To pursue advances in the detection utility of smart contracts, we consider the use of graph neural networks (GNNs) in deep learning to detect smart contract vulnerabilities [8], working alongside typical expert models given for vulnerabilities and adding the idea of detecting and blocking the operation of dangerous transactions at the EVM level to build a relatively complete set of smart contract vulnerability checking mechanisms with better detection and the ability to detect and block dangerous transactions at the EVM level to build a relatively complete set of smart contract vulnerability checking mechanisms with better detection and the ability to block risky behavior.

A. OUR APPROACH AND CONTRIBUTION

In this research, we present a smart contract vulnerability detection mechanism that detects vulnerabilities in smart contracts and suspends transactions of risky contracts after they are implemented and discovered to be flawed. Specifically, (1) The GNN model with improved detection efficiency and stronger generalization ability, as well as the expert model, collaborate to improve smart contract screening for specific vulnerabilities. This approach attempts to combine traditional expert rules with deep learning model methods to mitigate the interpretability concerns of deep learning methods while also compensating for expert rules' inability to detect vulnerabilities in the face of increasingly complex and large smart contracts. (2) We investigate and propose constraint rules for specific vulnerability types that, if the rules detect flaws with the contract, can stop contract transactions at the EVM level even after the smart contract has been deployed. This blocking feature is not limited to contract transactions, but can even encompass harmful behaviors within the EVM itself if the constraint constraints are properly defined. (3) To avoid wasting resources, the second phase of the vulnerability detection method only opens checks for high-value contracts identified by the user. (4) The approach also has a positive overall effect on the effectiveness of smart contract vulnerability identification, with an average score that is approximately 6 points higher than that of GNN model detection alone.

II. RELATED WORK

We talk about related work aimed at detecting smart contract vulnerabilities. For example, Slither [9], an open-source static analysis framework, works by taking the Solidity abstract syntax tree AST generated from the contract source code as initial input, converting the contract code into an intermediate representation SlithIR, which uses a static single assignment (SSA) form and a simplified instruction set to simplify the implementation of the analysis, while also preserving the semantic information lost when Solidity is converted to bytecode. EasyFlow [10], a method based on dynamic taint analysis for detecting overflow vulnerabilities in Ethernet smart contracts, also identifies against the SafeMath library and its various derivatives, effectively lowering the false alarm rate. HoneyBadger [11] is a smart contract automation platform that relies on symbolic execution [12] and well-defined heuristics. The first tool to detect EVM vulnerabilities using differential fuzzing techniques is EVMFuzzer [13], whose core idea is to continuously generate seed contracts and provide them to target and benchmark EVMs to find as many inconsistencies between execution results as possible, and eventually discover vulnerabilities through output cross-referencing.

Attempts have also been made to interpret contract code as text, assess its semantic and syntactic linkages, as well as control flow and data flow dependencies, and then implement smart contract vulnerability detection using appropriate learning models and detection targets. Eth2Vec [14] uses implicit knowledge to automatically learn the attributes of susceptible EVM bytecodes, which is enhanced by a neural network for natural language processing and can discover smart contract vulnerabilities by comparing the target EVM bytecode and the previously taught EVM bytecode. contractWard [15] balances the training dataset using two sampling methods, synthetic minority oversampling (SMOTE) and SMOTETOmek [16], and then detects vulnerabilities in smart contracts using five machine learning algorithms: eXtreme gradient enhancement (XGBoost) [17], adaptive augmentation (AdaBoost) [18], random forest (RF) [19], support vector machine (SVM) [20], and k-most-neighborly (KNN) [21].

III. PROBLEM STATEMENT

The smart contract vulnerability detection mechanism's work focuses on looking for these typical vulnerabilities.

A. REENTRANCY [22]

When a user contract calls another smart contract, if the called function is not found in the called contract or if the contract just receives tokens and no additional messages, it falls back to the fallback function. The reentry vulnerability frequently

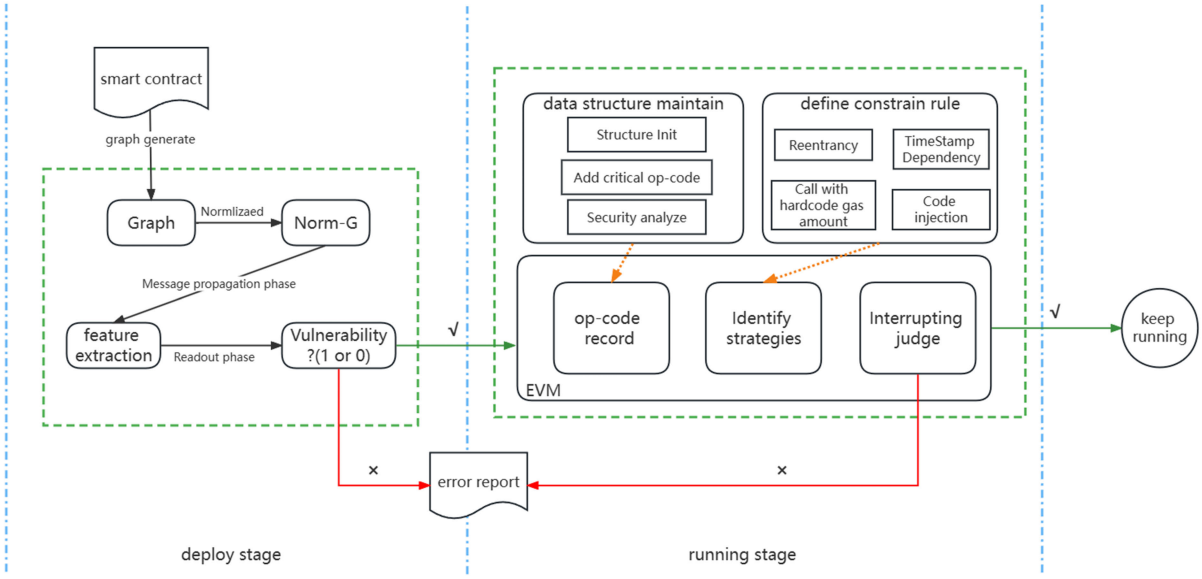


FIGURE 1. The overall process and function of the smart contract vulnerability detection mechanism.

occurs at this step of the fallback function, where an attacker can construct a special fallback function to attack the contract, for example, when the fallback function calls the transfer function in the victim contract again to transfer tokens to the attacking contract, then cycling through this operation can deplete the victim contract's balance. The infamous DAO assault, for example, exploited such weaknesses, resulting in the loss of more than $60 million in Ether.

B. CALL WITH HARDCODE GAS AMOUNT [22]

The transfer() and send() function in the message call forward a fixed amount (default 2300) of gas. Historical experience often suggests using these functions for value transfer to prevent re-entry attacks, but the cost of EVM's gas is subject to change during a hard fork, which can undermine contracts that have been deployed that make fixed assumptions about gas.

C. TIMESTAMP DEPENDENCY [22]

Each Ether block has a timestamp that can be read by a smart contract and is used as the foundation for some conditional judgments in some smart contracts. Sadly, while this is of little utility to the normal user, a powerful miner who can edit the timestamp within specific restrictions can change the timestamp's status and hence change the contract's output to his advantage.

D. CODE INJECTION [22]

The root cause of this vulnerability is the delegate call mechanism included in Ethernet, which allows contracts to execute code snippets from other contracts in their environment. If a malevolent user executes the code of an untrusted contract, he can easily intercept crucial information in the context of the attacked contract or even influence the victim contract to modify some important state variables for his objectives.

Details of the above vulnerabilities are also available from the **SWC** (https://swcregistry.io/).

IV. OUR APPROACH

As illustrated in FIGURE 1, our approach may be divided into two phases: before and after the deployment of a smart contract. Before deploying a contract, it is tested for vulnerabilities with a component that contains the GNN model, and if it is judged unqualified, the contract's deployment is halted and a bug report is generated and filed. During the runtime phase, the user must specify a value for the contract, and for high-value smart contracts, expert rules are used to analyze the sensitive opcodes involved in dangerous operations, interrupting this contract transaction once signs of a violation (i.e., violation of the defined constraint rules) are discovered and generating an error report for submission. When a smart contract passes through the second level of vulnerability testing without having its opcode execution halted, it continues to execute the code, guaranteeing that the vulnerability-detected smart contract transaction continues to operate normally.

A. GRAPH NEURAL NETWORK COMPONENTS

In general, graph-based vulnerability detection focuses on three aspects: how to describe the contract code as a graph, what method or algorithm to use to extract the graph's features, and finally, the selection of an appropriate model for label classification (i.e., vulnerability detection). The standard approach to describing the source code of a smart contract as a contract graph is to generate the graph based on the contract code's data flow and control flow relationships.

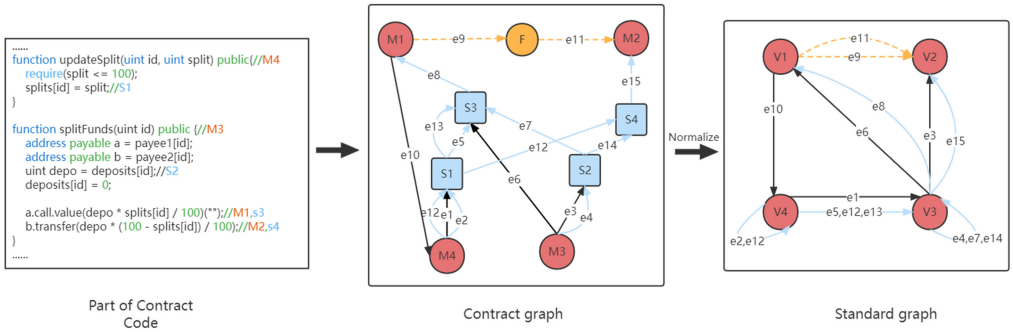


FIGURE 2. The generation and normalization of contract graphs. The second diagram, where M is the primary node, S is the secondary node, and F is the fallback node, shows the important function calls, key variables of the contract, and fallback functions in the contract.

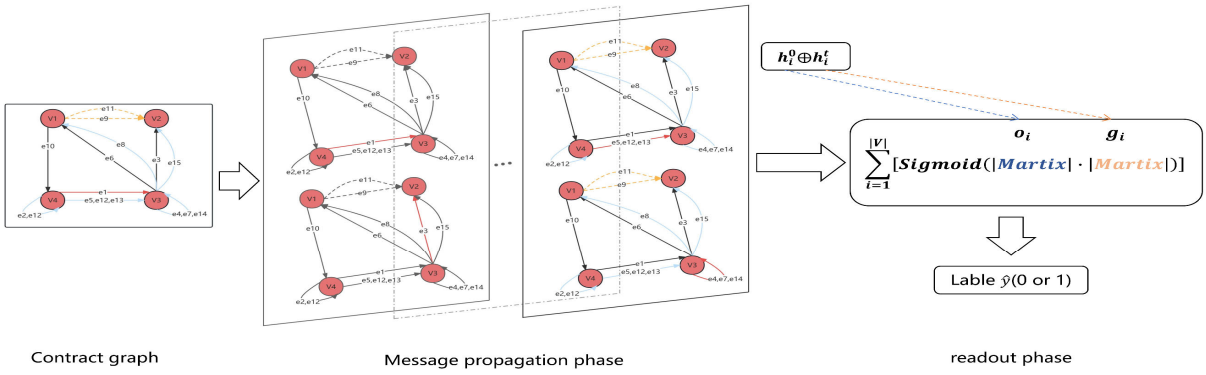


FIGURE 3. TMP network architecture.

The graph's nodes represent important function calls or key variables, and the edges represent changes in activity.

The study of graphs leads us to easily discover that most nodes of the contract graph play essentially the same role in the message delivery process, which averages each node's role in message propagation and fails to highlight the feature that some nodes play a more important role than others in actual message propagation. As a result, we devise a set of elimination rules to remove superfluous nodes from the generated contract graph to normalize it to a standard graph. The normalized graph is then processed using the temporal message propagation network model (TMP) [8], which computes the graph's global label and determines the presence of vulnerabilities based on the label.

FIGURE 2 depicts the standard graph generation process. The intercepted code is a contract code that contains a re-entrant vulnerability and payment transfer token functionality.

To begin, we must express the contract code as a contract graph with data and control relationships, where the updateSplit(), splitFunds(), call.value(), and transfer() functions are represented as M4, M3, M1, and M2 in the graph, and many key variables in the code also become S nodes in the graph, for the smart contract. The fallback mechanism (the fallback function is a special design of smart contracts and a dependency on which many vulnerabilities arise) requires the creation of a special fallback node to be reflected in the graph. The code's contextual relationship connects the nodes based on distinct information flows. The graph is then normalized by preserving only the major nodes and removing other nodes while aggregating the data flow and control flow information of the discarded nodes to the neighboring major nodes, resulting in the standard graph we want.

The TMP network includes two phases: message propagation and readout, which is our primary approach for extracting graph features and conducting label categorization, like FIGURE 3. TMP passes messages sequentially according to the temporal order of edges during the message propagation phase. TMP computes the labels of the complete graph G during the read-out phase by employing a read-out function that aggregates the final states of all nodes in G.

1) THE PHASE OF MESSAGE PROPAGATION

Messages are sent along edges, one at a time. The hidden state of each node h_i^0 is initialized at time step 0 using the properties of V_i nodes. The message propagates through the kth time edge e_k and updates the concealed state of V_{ek} (i.e., the end node of e_k) at time step k. As a result, the message m_k can be calculated using the hidden state of the starting node h_{sk}, and the edge type t_k.

$$x_k = h_{sk \oplus t_k} \tag{1}$$

$$m_k = W_k x_k + b_k \tag{2}$$

where $\oplus$ is the cascade operation, and W_k and b_k are network parameters.

2) THE READ-OUT PHASE

After traversing all of G's edges sequentially, the final hidden states of all nodes will be read out in order to calculate G's labels. Given that h_i^T represents the final state of the ith node, the projected label $\hat{y}$ can be calculated using the formula below.

$$s_i = h_i^T \oplus h_i^0 \tag{3}$$

$$g_i = softmax\left(W_g^{(2)}\left(\tanh\left(b_g^{(1)} + W_g^{(1)} s_i\right)\right) + b_g^{(2)}\right) \tag{4}$$

$$o_i = softmax\left(W_o^{(2)}\left(\tanh\left(b_o^{(1)} + W_o^{(1)} s_i\right)\right) + b_o^{(2)}\right) \tag{5}$$

$$\hat{y} = \sum_{i=1}^{|V|} Sigmoid\left(o_i \odot g_i\right) \tag{6}$$

where $\odot$ denotes the element-based product, and W and b with subscripts are the learning parameters of the model.

The TMP network must be fed a large number of standardized graphs generated by smart contracts and their actual vulnerability situation labels during the training phase, and the trained model is then used to add the normalized graphs and generate vulnerability detection labels. The first stage of the contract vulnerability testing mechanism is a graph neural network-based approach to finding flaws.

B. EXPERT KNOWLEDGE TO DETECT VULNERABILITIES

The main difference between this section and some existing traditional vulnerability detection methods is that even if a vulnerable smart contract is deployed after being fortunately ignored by the GNN component, vulnerability analysis can still be performed at the EVM level when the contract's suspicious opcode is about to be executed, and the operation can be interrupted in time when the opcode is found to be suspicious [23]. Using this method in the smart contract vulnerability detection system strengthens and completes our mechanism.

The opcode recording module, the vulnerability identification strategies detection module, and the interrupt module are all required for the above check. The opcode logging module must first initialize the data structure, then add suspicious sensitive code and examine it. Our unique vulnerability detection constraint criteria are used to construct the vulnerability identification module, which is based on a database of relevant vulnerabilities. When the code violates the rules, the interrupt module is triggered, the code transaction is terminated, and an error report is issued.

The vulnerability identification strategy is the most significant component of the expert knowledge detection part because its effectiveness is directly tied to the validation and blocking of risky transactions during the operation phase. We design relevant constraint rules to be responsible for verifying the security of contract transactions for several common vulnerability types in smart contracts after reviewing the empirical studies of various vulnerabilities in related literature [3], [22], [24], referring to expert models of these vulnerabilities [25], and based on the characteristics of some vulnerabilities: reentrant vulnerability, call with hardcode gas amount vulnerability, timestamp dependency vulnerability. The specific vulnerability detection methodologies are as follows.

1) REENTRANCY

Because re-entry attacks are usually used to steal tokens owned by smart contracts, we created numerous requirements to help detect contract reentrancy. (1) *iscall.value*, which examines if the function contains a call to *call.value* and, if so, identifies the contract code as a sensitive code of interest. (2) *BalanceDeduction*, which uses *call.value* to determine whether the user's balance is deducted following a transfer operation. This technique takes into account the possibility of code reentrancy by guaranteeing that the logic of modifying state variables occurs before permitting token transfers. To limit this, the following method is used:

$$\{A|acall : call, send; \exists\, isacall.value() > 0 \cap balanceDeduction() > 0\} \tag{7}$$

where A is the transaction conducted by the contract code and acall is the contract's value-related call. The danger of re-entry is determined when both requirements are met.

2) CALL WITH HARDCODE GAS AMOUNT

We believe that the main way to check for hard-coded gas quantity calls is to start with transfer() and send() and functions that specify a fixed amount of gas. When there is a send or transfer call in the contract, and there exists a function that assigns the amount of gas as a left value, it is considered that there is a hard-coded number of gas call vulnerabilities. The strategy is as follows:

$$\{A|istransfer() \cup issend() \cup isSetgas()\} \tag{8}$$

where A is the contract-run transaction, isSetgas() looks for the gas keyword in the contract and checks whether there is a function for its fixed assignment operation. Satisfying any of the above conditions are considered to have a hard-coded gas quantity call vulnerability.

3) TIMESTAMP DEPENDENCY

It is established that a smart contract has a timestamp dependency vulnerability when it uses the value of a *block.timestamp* is one of the prerequisites for completing operations like balance modifications and transfers. As a result, three conditions are established for testing this vulnerability: (1) Determine whether the code contains a call to the operation code *block.Timestamp*, and if so, classify the contract code segment as of interest. (2) Determine if the call allocates the value of *block.Timestamp* to another variable or passes it directly as an argument to another function. (3) Confirm that the call to *block.Timestamp* is not used as a prerequisite for some operations involving the return result of a contract transaction.

$$\{(A|isblock.Timestamp() \cap \exists V = value(block.Timestamp) or$$

$$\exists F(value(block.Timestamp)) \cap istimestamp\ Contamination())\} \quad (9)$$

where A signifies the contract transaction, V is the variable in the contract code, F is the contract function, and *value()* is the value of the object in parenthesis. For the third condition, we create a sub-pattern called *istimestampContamination*, which uses an approach similar to taint analysis to determine whether the timestamp value has contaminated the transaction result and returns a bool type. If all three of the contract's features are met, it is assumed that there is a timestamp dependence vulnerability.

4) CODE INJECTION

The major check for code injection risk is to determine whether there is an untrusted delegate call, so first confirm if there is a delegate call during contract transaction execution, and if so, mark that contract code as of interest. Then, determine whether the call parameters are related to the root contract return value. If it is relevant, Delegatecall may call malicious code to change some key values of the root contract. The following is the specific strategy:

$$\{A|isDelegatecall() \cap rootOutput.relevent(currF()) > 0\} \quad (10)$$

where A signifies the contract transaction, *rootOutput* denotes the parameter associated with the return value of the root contract's function, and *currF* denotes the operation executed by the other contract's code delegated to the call. If all of the aforementioned conditions are met, the contract is considered vulnerable to malicious code injection.

We created an automated tool to validate contract functions automatically by recording opcodes based on keyword matches in smart contract functions and including the intended vulnerability identification approach.

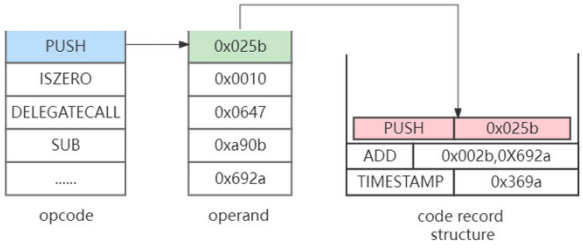


FIGURE 4. The basic process of opcode pressing into the stack.

The various detection strategies outlined above necessitate the maintenance of a data space in which the code structure of interest is kept and the strategy analysis and decision to interrupt the contract are performed. We chose to implement this space using a stack structure because, in general, the memory used to run the code of a smart contract is also a stack structure, and using a stack to accomplish the recording of opcodes is consistent with programming intuition on the one hand, and has the advantage of being simple to operate and maintain on the other. FIGURE 4 depicts the fundamental steps for recording opcodes.

The data space utilized to analyze opcodes is depicted in FIGURE 4. The code record stack's elements are classified as opcodes and operands. For example, if the current opcode is PUSH and the operand is $0 \times 025b$, the code record stack will push both into the stack simultaneously for run-time vulnerability analysis.

Algorithm 1 : Core Control Process

```
Input: IS: Identify strategies.
Function main():
  if high_value() then
    instrument(IS);
  else
    execute(code);
  end
End Function
Function instrument(strategie IS):
   oplist = [];
   op_Stk = new STACK();
   for code in opcodes do
     if code.isCALL() then
       if !op_Stk.isEmpty() then
          oplist.append(op_Stk);
          op_Stk.clear();
          op_Stk.push(new Target
          (CALL,code.operands()));
       end
     end
     else if code.isInteresting()then
      item = code.name();
      op_Stk.push(new Target(item,code.operands()));
      for constrain in IS do
        if !Test(constrain, oplist, op_Stk) then
           Interrupt();
        end
      end
   end
   execute(code);
end
End Function
```

At the virtual machine level, the fundamental control process algorithm incorporates vulnerability identification policies and interruption methods. Algorithm1 depicts the main control procedure.

Smart contracts that advance to this round in the vulnerability detection procedure are checked by the GNN component and can be considered initially secure for the time being. Due to the time overhead pressure of the contract transaction, as well as to minimize excessive waste of resources, the main function is configured to allow the user-identified low-value contracts to bypass the expert knowledge of vulnerability detection, while the user-highly valued contract transactions to carry out the constraint rules analysis.

By establishing a stack op_Stk to store the opcodes before detection, the specific detection policy IS is utilized as input

TABLE 1. Comparison of the effectiveness of DM with other networks for vulnerability detection.

Methods		Vanilla-RNN[26]	LSTM[27]	GRU[28]	GCN[29]	DM
Reentrancy	Acc(%)	49.64	53.68	54.54	77.85	**89.74**
	Precision(%)	49.82	51.65	53.10	70.02	**85.35**
	Recall(%)	58.78	67.82	71.30	78.79	**86.19**
	F1(%)	50.71	58.64	60.87	74.15	**85.76**
Timestamp dependency	Acc(%)	49.77	50.79	52.06	74.21	**88.52**
	Precision(%)	51.91	50.32	49.41	68.35	**82.07**
	Recall(%)	44.59	59.23	59.91	75.97	**86.23**
	F1(%)	45.62	54.41	54.15	71.96	**84.10**
Code injection	Acc(%)	49.12	51.98	53.74	72.98	**88.62**
	Precision(%)	42.64	50.64	52.01	69.82	**83.69**
	Recall(%)	47.55	63.47	61.64	76.84	**87.57**
	F1(%)	44.96	56.33	56.41	73.16	**85.58**
Call with hardcode gas amount	Acc(%)	52.12	55.28	57.74	74.92	**90.62**
	Precision(%)	48.64	51.91	54.01	71.03	**86.69**
	Recall(%)	49.55	68.47	63.64	77.97	**87.99**
	F1(%)	49.09	59.05	58.43	74.34	**87.34**

to the staking function. It is vital to remember that each CALL opcode for a piece of code indicates that there is a sequence of operations within that call that belong to the same call, and for analysis reasons, a delimited group of all opcodes for this call is required. When an opcode is a CALL, if op_Stk is not empty, add its contents to the opcode list, else empty op_Stk and press the CALL with its operands into op_Stk. Furthermore, if the current opcode is sensitive, as described in lines 22 to 24, the vulnerability checking mechanism will test it, in turn, using the defined constraint criteria, and if it fails, the contract will be halted; otherwise, the code will continue to run.

V. EXPERIMENT

We assess the approach provided in this paper using the Ethereum smart contract dataset in this part. And we attempt to answer the following questions using the vulnerability detection mechanism provided:

- Q1: Is the proposed vulnerability detection mechanism capable of discovering flaws? Can it stop the execution of risky contracts?
- Q2: Does the improved vulnerability detection approach provided in this paper improve the effectiveness of vulnerability protection?
- Q3: How does the mechanism stack up against other methods?

A. EXPERIMENTAL SETUP

1) DATASETS

307396 functions from 40932 smart contracts are included in the Ethernet Smart Contracts dataset (ESC). 5013 functions in this dataset include at least one call statement to call.value, making them potentially vulnerable to reentrant vulnerabilities. There are 4833 routines with BLOCK.TIMESTAMP statements that could result in a timestamp dependence issue. 6896 routines employ the DELEGATECALL command at least once, indicating the presence of a code injection vulnerability.

2) SETTINGS

All of the following trials were carried out using a PC outfitted with a 3.2 GHz AMD Ryzen7 CPU, a 1660 Ti graphics processor, and 16 GB of RAM. For each dataset, we randomly selected 80% of the functions as the training set and the remaining 20% as the test set, ran several experiments and reported the average results. We also chose over 1000 smart contracts with each of the four different vulnerabilities as direct input to the second phase of the detection mechanism, in which each function in the contract is randomly executed at least once with arbitrary parameters to see if the vulnerability security mechanism can react to block dangerous transactions. For ease of addressing, in this section we will refer to the smart contract vulnerability detection mechanism as DM, the first stage of the detection mechanism is called DM-1, and the second stage of the detection mechanism is called DM-2.

B. EXPERIMENTAL RESULTS

1) COMPARISON WITH OTHER NETWORK ALGORITHMS

We compare DM to various network algorithms for contract code reentrancy, timestamp dependency, and malicious code injection. TABLE 1 compares various methodologies for each vulnerability type, with Acc, Precision, Recall, and F1 score as the primary metrics. Before understanding these metrics we need to explain TP, FP, TN, and FN.

TP(True Positive): A correct positive example, where an instance is a positive class and is also determined to be a positive class.

FP(False Positive): A wrong positive example, false alarm, originally a false class but judged as the positive class.

TN(True Negative): For the correct counterexample, an instance is a false class and is also determined to be a false class.

FN(False Negative): A wrong counterexample, omission, positive class but decided as a false class.

While **Acc** = (TP+TN)/(TP+TN+FP+FN) counts how many of all predicted results are predicted correctly, **Precision** = TP/(TP+FP) counts how many of all predicted results are positive and correct, which means how many are true positives, **Reacll** = TP/(TP+FN) counting the number of correct predictions among all the actual categories that are positive, **F1** = (2×Precision×Recall)/(Precision+Recall) is the summed mean value of precision and recall. In summary, we believe that these four indicators can largely reflect the effectiveness of the detection method for vulnerability detection.

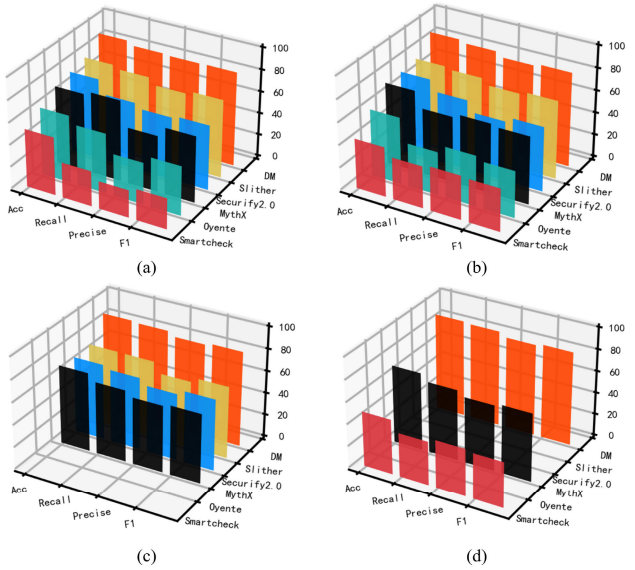


FIGURE 5. (a) represents the detection effect of reentrancy, (b) represents the detection effect of timestamp dependency, (c) represents the detection effect of code injection, (d) represents the detection effect of call with hardcode gas amount.

Vanilla-RNN [26]: a two-layer recurrent neural network that takes a sequence of codes as input and updates its hidden state by recursion.

LSTM [27]: a widely used recurrent neural network that periodically updates the unit state when sequences of codes are read continuously.

GRU [28]: gated loop unit that uses a gating mechanism to process code sequences.

GCN [29]: takes the graph as input and performs a layered convolution of the graph using Laplacian.

TABLE 1 demonstrates that our smart contract vulnerability detection mechanism has a good detection effect for the three listed vulnerabilities; even for the GCN technique, which has the best effect among other methods, there is an average overall increase of more than 10 points; we reasonably infer that this is because the traditional expert knowledge supplied in DM-2 has corrected certain misjudgment of contract vulnerabilities by the GNN model in MD-1, and so improved detection.

2) COMPARISON WITH OTHER DETECTION TOOLS

For reentrant and timestamp-dependent vulnerabilities, we compared the vulnerability detection mechanism to some current detection technologies. SmartCheck [30], Oyente [5], MythX, Securify2.0, and Slither [9] are among the detection technologies that have taken part. FIGURE 5 depicts the outcomes of the comparison.

Our method is unquestionably superior to some current vulnerability tools in the inspection of re-entry vulnerabilities and timestamp-dependent vulnerabilities because most existing inspection tools are based on expert models and then apply a variety of detection tools such as taint analysis, static frame analysis, and so on, whereas DM not only relies on expert knowledge but also adds GNN to the vulnerability.

3) ABLATION TESTS

To highlight the distinction between EVMs with DM-2 and ordinary EVMs, we test smart contracts with each of the four vulnerability categories to see if they can react to block risky transactions, and the results are displayed in TABLE 2.

TABLE 2. Comparison of DM-2 and ordinary EVM in stopping dangerous behaviors.

Vulnerability Type	Number of contract	Ordinary EVM	MD-2
Reentrancy	200	0	176
Timestamp dependency	200	0	172
Code injection	200	0	178
Call with hardcode gas amount	300	0	269

We improve the detection approach in other work by incorporating expert knowledge into the deep learning model to correct the findings and by developing a graph normalization process for the deep learning GNN model to increase the effect of feature extraction. To test how well the vulnerability detection works, we delete the expert knowledge module and the normalization phase of the graph.

In general, we utilize the F1 score to indicate the detection effect demonstrated for distinct vulnerabilities, and compare the DM with the part of the corresponding module deleted, as shown in FIGURE 6.

When evaluating vulnerability detection performance, we choose the detection method that takes fewer learning rounds and higher accuracy when the accuracy of each component is fitted to work better. FIGURE 7 depicts the experimental outcomes.

C. EVALUATION AND ANALYSIS

TABLE 2 shows that with DM-2, EVM detects and blocks 176 contracts for every 200 contracts with reentry vulnerabilities, 172 contracts for every 200 contracts with timestamp

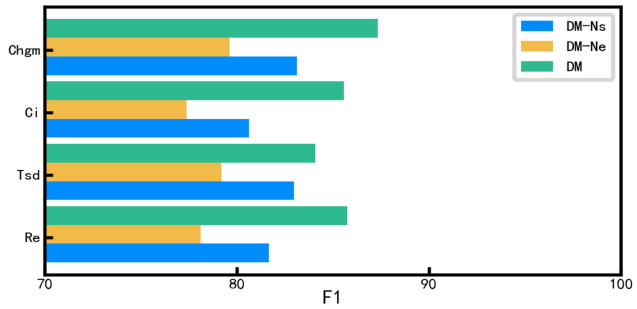


FIGURE 6. Comparison of ablation test results. The detection mechanism that removes the expert knowledge module is denoted by DM-Ns, the detection mechanism that removes the graph normalization process is denoted by DM-Ne, the code injection vulnerability is denoted by Ci, the timestamp dependency vulnerability is denoted by Tsd, and the re-entry vulnerability is denoted by Re, the call with hardcode gas amount is denoted by Chgm.

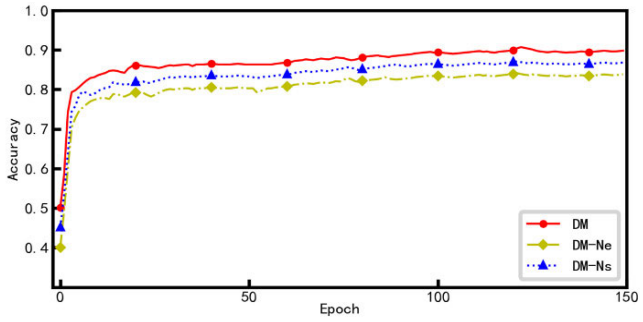


FIGURE 7. Ablation test performance comparison.

dependency vulnerabilities, 178 contracts for every 200 contracts with code injection risks, and 300 contracts for call with hardcode gas amount vulnerabilities, detecting and blocking 269 contracts with problem. This shows that for Q1, the security mechanism has a good probability of detecting these types of vulnerabilities and successfully blocking the dangerous behavior of these contracts. This is since our approach embeds a vulnerability detection control process in EVM that is bound by expert rules to detect vulnerabilities and prevent harmful behaviors at the VM level

The ablation experiment addresses Q2, and TABLE 2 shows that EVMs without DM-2 cannot prevent a susceptible contract from running. FIGURE 6 shows the impact of the trials with the expert rule detection module and the graph normalization procedure. For either vulnerability, the entire DM is more effective than the missing DM-Ne or DM-Ns, and the F1 score declines more when the expert knowledge detection module is removed. This demonstrates the significance of the expert knowledge module, which is more complementary to the vulnerability detection mechanism than the graph normalization procedure. FIGURE 7 shows the detection performance for DM, DM-Ns, and DM-Ne, and it is apparent that the total DM is superior. We attribute DM's superior performance to the fact that DM-Ne does not evaluate established security patterns and ignores important variables. The graph normalization procedure enhances performance significantly because the normalized graph keeps only the main nodes relevant to vulnerability identification, compressing the wandering routes during graph embedding and minimizing the likelihood of nodeless participation, which dramatically improves feature extraction efficiency and hence DM performance. These findings highlight the importance and efficacy of GNN models with previous graph normalization when linked with expert knowledge.

The comparison between DM and other methods is presented in TABLE 1 and FIGURE 5, and we can see that DM has a significant improvement in vulnerability identification compared to other network models, while GCN is the best among other methods. We anticipate that the other networks will lose valuable information from the smart contract code when processing it, because they neglect structural information from the contract program, such as data flow and call linkages. This implies that viewing source programs as code sequences blindly is not appropriate for vulnerability detection tasks, and that representing source code as graphs and applying graph neural networks is more promising. According to the graph, DM also has a deep learning additive advantage over other traditional detection tools. This data precisely answer Q3, and our DM still has some improvement and advantage over other detection methods and instruments.

VI. SUMMARIZE

In this paper, we study the work of combining GNN models with expert models working at the EVM level for smart contract vulnerability detection. The smart contract vulnerability detection technique presented in this work improves both its detection capabilities as a vulnerability detection tool and its capacity to function as a stopper for contracts performing risky operations. Although the time overhead will be more than the EVM without the second phase of the protection mechanism, the mechanism provides the user with the option of whether or not to activate the second phase of the protection mechanism to prevent wasting resources like time and gas. We believe that the work in this paper will play a significant role in the future development of a more secure and reliable smart contract environment.

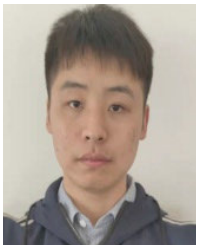
Improving smart contract vulnerability detection technologies and associated algorithms will be a key research focus in the future. Future research will consider, on the one hand, improving the effectiveness of GNN's aggregation algorithm or better-generating contract graphs to capture the semantic information of contract code, and on the other, exploring more vulnerability identification strategies to cover more types of risks or reducing the time overhead of running the system by optimizing the spatial structure of the core control flow and opcode records.

REFERENCES

[1] J. Cheng, "Current status and prospects of blockchain technology," in *Proc. Int. Conf. Artif. Intell. Secur.* Singapore: Springer, 2020, pp. 674–684.
[2] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2019, pp. 347–356.
[3] Y. Ni, C. Zhang, and T. Yin, "A review of research on smart contract security vulnerabilities," *J. Inf. Secur.*, vol. 5, no. 3, pp. 78–99, 2020.

[4] V. Buterin, "A next-generation smart contract and decentralized application platform," *White Paper*, vol. 3, no. 37, p. 37, 2014.
[5] S. Badruddoja, R. Dantu, Y. He, K. Upadhayay, and M. Thompson, "Making smart contracts smarter," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency (ICBC)*, May 2021, pp. 1–3.
[6] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 67–82.
[7] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, "Evaluating explanation methods for deep learning in security," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Sep. 2020, pp. 158–174.
[8] Y. Zhuang, "Smart contract vulnerability detection using graph neural network," in *Proc. IJCAI*, 2020, pp. 3283–3290.
[9] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.
[10] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "EASYFLOW: Keep Ethereum away from overflow," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, May 2019, pp. 23–26.
[11] C. F. Torres and M. Steichen, "The art of the scam: Demystifying honeypots in Ethereum smart contracts," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)* Santa Clara, CA, USA: USENIX Assoc., 2019, pp. 1591–1607.
[12] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, May 2019.
[13] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "EVMFuzzer: Detect EVM vulnerabilities via fuzz testing," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 1110–1114.
[14] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2 Vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts," in *Proc. 3rd ACM Int. Symp. Blockchain Secure Crit. Infrastruct.*, May 2021, pp. 47–59.
[15] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated vulnerability detection models for Ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, Apr. 2021.
[16] K. Bhargavan, "Formal verification of smart contracts: Short paper," in *Proc. ACM workshop Program. Lang. Anal. Secur.*, 2016, pp. 91–96.
[17] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 785–794.
[18] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, Aug. 1997.
[19] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[20] J. A. K. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Process. Lett.*, vol. 9, no. 3, pp. 293–300, Jun. 1999.
[21] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp. 21–27, Jan. 1967.
[22] Y. Zhang and J. Ma, "Overview of vulnerability detection methods for Ethereum solidity smart contracts," *Comput. Sci.*, vol. 49, no. 3, pp. 52–61, 2022.
[23] F. Ma, M. Ren, Y. Fu, M. Wang, H. Li, H. Song, and Y. Jiang, "Security reinforcement for Ethereum virtual machine," *Inf. Process. Manage.*, vol. 58, no. 4, Jul. 2021, Art. no. 102565.
[24] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in Ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.
[25] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 2, pp. 1296–1310, Feb. 2023.
[26] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proc. Int. Conf. Neural Netw. (ICNN)*, Jun. 1996, pp. 347–352.
[27] T. Zia and U. Zahid, "Long short-term memory recurrent neural network architectures for Urdu acoustic modeling," *Int. J. Speech Technol.*, vol. 22, no. 1, pp. 21–30, Mar. 2019.
[28] A. A. Ballakur and A. Arya, "Empirical evaluation of gated recurrent neural network architectures in aviation delay prediction," in *Proc. 5th Int. Conf. Comput., Commun. Secur. (ICCCS)*, Oct. 2020, pp. 1–7.
[29] S. Fu, W. Liu, D. Tao, Y. Zhou, and L. Nie, "HesGCN: Hessian graph convolutional networks for semi-supervised classification," *Inf. Sci.*, vol. 514, pp. 484–498, Apr. 2020.
[30] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2018, pp. 9–16.

ZHENPENG LIU received the B.S. and M.S. degrees from the School of Cyberspace Security and Computer, Hebei University, and the Ph.D. degree from Tianjin University. He is currently a Professor with the School of Cyberspace Security and Computer, Hebei University. His research interests include network information security and outlier detection.



MINGXIAO JIANG is currently pursuing the M.S. degree with the School of Cyberspace Security and Computer, Hebei University, Baoding, Hebei. His research interests include network information security and data integrity verification.

SHENGCONG ZHANG is currently pursuing the M.S. degree with the School of Cyberspace Security and Computer, Hebei University, Baoding, Hebei. His research interests include deep learning, graph embedding, and natural language processing.

JIALIANG ZHANG is currently pursuing the M.S. degree with the School of Cyberspace Security and Computer, Hebei University, Baoding, Hebei. His research interests include the Internet of Things, network security, and deep learning.

YI LIU received the M.S. degree from the School of Cyberspace Security and Computer, Hebei University, Baoding, Hebei. He is currently an Engineer with the Information Technology Center, Hebei University. His research interests include cloud computer security, privacy protection, and data integrity verification.