


ESA: An Enhanced Sequential Algorithm-Based Model for Smart Contract Vulnerability Detection


Jinghan Liu

(Henan University, Kaifeng, China)

 <https://orcid.org/0009-0001-0180-1120>, liujinghan002@163.com)


Hui Zhao

(Henan University, Kaifeng, China)

 <https://orcid.org/0000-0002-6826-1026>, zhz@henu.edu.cn)


Chenyang Lin

(Henan University, Kaifeng, China)

 <https://orcid.org/0009-0008-6084-3057>, 1345789663@qq.com)


Dan Wang

(Henan University, Kaifeng, China)

 <https://orcid.org/0009-0006-4501-4299>, wdan00@163.com)

Shufan Li

(Henan University, Kaifeng, China)

 <https://orcid.org/0009-0003-9754-5567>, ysylsfshangan@163.com)

Abstract: The current security problem of smart contracts is becoming a common concern for researchers and developers. Existing smart contract vulnerability detection methods rely heavily on fixed expert rules, resulting in low detection accuracy. In order to cope with complex and changing smart contract application scenarios, we chose to use graph neural networks to detect vulnerabilities. In this paper, we proposed a vulnerability detection model called ESA based on the enhanced sequential algorithm. During the coding process, the contract function source code is described as a contract graph, which increases the model's global insight into node features during the learning process and reduces the number of noise nodes unrelated to vulnerabilities while retaining sufficient contextual semantic features. Compared to the cutting-edge methods, our model has significantly improved the accuracy of reentrant and timestamp dependency vulnerabilities, with detection accuracies of 89.09% and 88.49%, respectively.

Keywords: Blockchain, Smart contract, Vulnerability detection, Graph neural network, Attention mechanisms

Categories: I.2

DOI: 10.3897/jucs.137000

1 Introduction

Blockchain is an emerging technology that has gained significant traction in recent years. The primary advantage of blockchain is that it enables the creation of an irreversible transaction history on a distributed ledger, facilitating the purchase of items using digital

currency without needing third-party processing and authentication. Three key elements of blockchain technology are distributed ledger technology with an immutable record, consensus algorithms, and smart contracts. The distributed ledger technology and consensus algorithms that comprise the Ethereum platform serve to enhance its security. In turn, the smart contracts that are integral to the Ethereum platform facilitate the execution of operations that manage associated assets. A smart contract is a contract that is written in code and automatically executed without the need for a third party to manage or execute it. A multitude of applications, including those in logistics, insurance, medicine, energy, and non-alternative currencies [Zhao et al. 2021, Fan et al. 2019], are employing the use of smart contracts.

The majority of smart contracts are written in Turing-complete programming languages, most notably Solidity. These languages enable developers to define and execute complex business logic embedded in the blockchain network, thereby facilitating tamper-proof and verifiable contract execution. The operation of smart contracts is based on a network of nodes on the blockchain, which ensures that each participant can verify the results of the contract's execution. This enhances trust and transparency.

While smart contracts have the potential to provide highly automated, transparent, and programmable transaction execution, their security is also a significant challenge. The immutability of smart contracts introduces a significant risk of vulnerability, which has the potential to negatively impact both the user experience and the credibility of blockchain smart contracts. In 2016, a reentrant vulnerability was exploited in the DAO (Decentralized Autonomous Organizations) [Meher et al. 2019], resulting in the theft of over 3.6 million Ether, valued at over 70 million dollars. This theft directly led to the loss of over 3.6 million Ether user accounts. Consequently, the research direction of smart contract vulnerability detection is highly valuable, and it is imperative to develop an efficacious tool for smart contract vulnerability detection.

In previous research, the majority of smart contract vulnerability detection methods employed static analysis to search for pre-defined vulnerability patterns, which are typically identified by experts, to detect target smart contracts. The manually defined vulnerability patterns are relatively simple and therefore cannot cover all complex vulnerability patterns, which leads to a high underreporting rate. Furthermore, the exponential growth of smart contracts has made it increasingly challenging for a small number of experts to design precise vulnerability patterns. Researchers have proposed various machine learning-based vulnerability detection methods for smart contracts, intending to reduce the reliance on expert-defined patterns. Some works consider smart contracts to be natural language and represent them as flat sequences, after which sequential neural networks (e.g., LSTM [Sak et al. 2014] or RNN [Goller and Kuchler 1996]) are employed for feature learning and vulnerability detection. The methodologies mentioned above either solely consider the textual characteristics of operational code or construct semantics and control information flow at the source code level. Nevertheless, the logic and structural information embedded within smart contracts is more intricate than those in natural language. These approaches consider only the textual features of operational code, which may result in the loss of subtle semantic features within the code structure, such as data or control dependencies. This can lead to inaccuracies in the detection of vulnerabilities.

In contrast to conventional neural network models, which typically utilize vectors or matrices as inputs, Graph Neural Network (GNN) employs graph structures as inputs. The fundamental concept of GNN is to model the interaction between nodes as an information transfer process, thereby obtaining the global graph structure features. This feature allows GNN to perform forward and backward propagation for graphs of arbitrary

size and shape, rendering it suitable for processing data without fixed structures like smart contract source code. At each node, the GNN combines the features of that node and its neighboring nodes into a new feature vector, which is then transmitted to the subsequent node. This process may be repeated several times until a termination condition is met. In previous studies, researchers have not explored the temporal features of smart contracts in more depth. In this paper, we focus on a specific smart contract vulnerability task and explicitly consider the different roles and sequential relationships of the program elements, not only focusing on the aggregation of features of the nodes but also increasing the degree of attention paid to the temporal characteristics of the edges of the contract graph. Consequently, we have elected to employ GNN with this distinctive architectural configuration in the development of our model for vulnerability detection by constructing the source code as a contract graph, which maximizes the preservation of the syntactic and semantic features of the code.

Current mainstream graph construction methods organize (i) syntactic features as tree structures and embody them in Abstract Syntax Trees [Mou et al. 2016, Zhang et al. 2019] (AST), (ii) semantic features are embodied in relations between statements and represented as graph structures, e.g., Control Flow Graphs [Phan et al. 2017] (CFG), and (iii) structural features are modeled as nodes using Code Property Graphs [Suneja et al. 2020, Yamaguchi et al. 2014] (CPG), and the control flows between statements are represented as edges. It is important to note that the tree structure in AST is progressively hierarchical, with the root of the tree representing the leaf nodes. In order to unify different heterogeneous features into a graph structure, it is necessary to learn the features by aggregating the neighbor information from each node. However, this operation does not consider whether the original leaf-to-root order is appropriate, and some semantic features are lost. The core of CPG is an attribute-based graph model, where each node and edge can carry multiple attributes, which can store metadata or other analysis information. Given the complexity of CPG, the process of modeling the source code of smart contracts will result in the accumulation of excessive redundant information, which will subsequently be transformed into noise within the model, thereby impairing the model's ability to detect patterns effectively.

Smart contracts possess complex control flow and data flow information. In consideration of the scenario above, we have elected to utilize the control flow graph approach for the modeling of the contract graph. In comparison to the conventional control flow graph approach, the approach proposed in this thesis emphasizes that the nodes possess varying degrees of importance to one another. Our proposed model is capable of fully characterizing the rich semantic information present within the source code. In describing the source code of the contract function as a contract graph, we retain sufficient contextual semantic features while reducing noisy nodes unrelated to the vulnerability. Furthermore, we optimize the model by adding an enhanced sequential attention algorithm. Upon analysis of the experimental results, it can be seen that the model proposed in this paper has high accuracy in detecting vulnerabilities.

The primary contributions of this paper are as follows:

- We proposed a novel vulnerability detection model, ESA, which comprehensively considers the semantic interrelationships of smart contract contexts and incorporates the smart contract fallback function into the construction of the contract flow graph while minimizing the influence of noise that is not pertinent to vulnerability detection.
- In the construction of the contract graph, we proposed a new rule for establishing edges, which assigns varying priorities to different types of edges. This rule is applied to our proposed timing-enhanced attention mechanism algorithm.
- We conducted experiments on a dataset of common vulnerability types and com-

pared our method with traditional smart contract vulnerability detection tools and current mainstream deep learning methods. With regard to the metrics of accuracy, recall, precision, and F1 value performance, our method evinces a notable enhancement in its efficacy.

The remainder of this paper is organized as follows: Section 2 presents a review of related work. Section 3 provides an explanation of the rationale behind the development of our method. Section 4 offers a detailed account of the implementation of our method. Section 5 presents the experimental setup of the model and an analysis of the experimental results. Section 6 offers a summary of the paper and suggests avenues for future research.

2 Related work

2.1 Traditional Detection Methods

In recent years, four main traditional methods for detecting vulnerabilities in smart contracts have emerged. These include symbolic execution, formal verification, intermediate representation, and fuzzy testing.

Symbolic execution is a method for analyzing software that treats program variables as symbols and systematically explores all possible execution paths. It is a powerful technique for analyzing software behavior, but its effectiveness depends on the size and complexity of the program being analyzed. Examples of tools that employ symbolic execution for vulnerability detection include Oyente [Luu et al. 2016], Securify [Tsankov et al. 2018], and Mythril [Mueller 2017].

Formal verification is a crucial technique for ensuring the security of smart contracts. By employing formal language, the concepts, judgments, and reasoning inherent in the contract are transformed into formal models, eliminating ambiguities and non-universality. Subsequently, the correctness and security of the functions within the smart contract are rigorously verified by applying rigorous logic and proofs. Famous formal verification methods include F* framework [Grishchenko et al. 2018], KEVM framework [Hildenbrandt et al. 2018] and ZEUS [Kalra et al. 2018].

Intermediate representations are used to convert smart contract source code into a form that is easier to analyze. Nevertheless, this approach may result in a loss of precision when analyzing the source code. Slither [Feist et al. 2019] converts smart contract Solidity source code into an intermediate representation of SlithIR, which employs a static single allocation form and a streamlined instruction set to simplify the process of contract analysis. Vandal [Brent et al. 2018] converts bytecode into a higher-level intermediate representation in the form of logical relationships and then uses a novel logic-driven approach to detect contract vulnerabilities. SmartCheck [Tikhomirov et al. 2018] converts smart contract Solidity source code into an XML-based intermediate representation and then utilizes XPath patterns to detect smart contract vulnerabilities.

Fuzzy testing entails the generation of input data at random, thereby simulating a variety of scenarios that are employed to assess the security of smart contracts. However, the efficacy of the fuzzy test is contingent upon the quality of the input generator. ContractFuzzer [Jiang et al. 2018] represents the inaugural dynamic analysis method for Ethereum smart contract security vulnerabilities based on fuzzy testing. It generates test cases based on the ABI specification of the smart contract and defines test schemes for the purpose of detecting security vulnerabilities. Regurad [Liu et al. 2018] is a fuzzy test analyzer that is specifically designed to identify reentrant vulnerabilities in smart contracts. By generating a multitude of random and diverse test cases iteratively, it

performs fuzzy tests on smart contracts, thereby tracking the execution of contracts and further dynamically identifying reentrant vulnerabilities. ILF [He et al. 2019] is a fuzzy test for smart contracts based on a neural network. The symbolic execution engine is employed to generate an effective test and call sequence, which serves to guide the feature learning of the neural network model and thereby facilitate effective vulnerability detection.

2.2 Detection Methods Based on Neural Network

The advent of machine learning and deep learning technologies has enabled the development of a multitude of sophisticated security detection methods. In the context of novel types of security vulnerabilities, neural network-based detection methods demonstrate notable scalability and adaptability, facilitating more effective detection of contractual vulnerabilities while simultaneously enhancing the speed and accuracy of the detection process.

In a recent study, [Yu et al. 2021] proposed a sophisticated deep learning-based system framework to detect smart contract vulnerabilities automatically. This framework, called DeeSCVHunter, focuses on two types of smart contract vulnerabilities: reentrancy and time dependency. Additionally, the authors proposed a new concept, Vulnerability Candidate Slices (VCSs), to help the model capture the critical points of vulnerabilities. [Cai et al. 2023] constructed graphical representations of smart contract functions with syntactic and semantic features by combining Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependency Graph (PDG). Subsequently, program slicing is conducted to normalize the graph and eliminate superfluous information that is not pertinent to the identification of vulnerabilities. Finally, a bidirectional gated graph neural network model with hybrid attention pooling is employed to identify potential vulnerabilities in smart contract functions. [He et al. 2023] proposed an extended SAGConv and Topkpooling graph neural network (ST-GNN) to learn the features of each node in the tree. To enhance the precision of the detection process, some non-critical nodes are removed and consolidated to accentuate the critical nodes and the sequence of operations. This approach culminates in the development of a novel vulnerability detection model, GraphSA. [Zhang and Liu 2022] employ two distinct training models, namely Graph Neural Network (GNN) and Graph Matching Network (GMN), to construct two types of vulnerability detection models, ASGVulDetector and BASGVulDetector, for static analysis. These models are designed to identify vulnerabilities in Ethereum smart contracts from two distinct perspectives: the source code and the bytecode. [Zhen et al. 2024] put forth a method for identifying vulnerabilities in smart contracts, namely DA-GNN. This model employs a dual-attention mechanism that incorporates both the node semantic features and the relational features between nodes into the GAT, thereby enabling node embedding updates.

2.3 Graph Neural Network

Graph Neural Network (GNN) is a deep learning framework that has emerged in recent years and is capable of learning directly from graph-structured data [Scarselli et al. 2008]. In the context of non-Euclidean data, GNN is capable of capturing more profound levels of information with greater precision and accuracy than other deep learning techniques. For software programs, such as code and smart contracts, processing using graph structuring methods not only preserves the overall program structure but also

expresses the semantic and syntactic information of the program. The acquisition of richer data features through the use of graph neural network methods for learning and reasoning about graph structures is a significant improvement over text sequence-based representation and learning methods, particularly in terms of processing accuracy for relevant programs.

A number of variants of graph neural networks already exist. To illustrate, the Graph Convolutional Network (GCN) aggregates the features and labels of the central node and its neighboring nodes through a multilayer graph convolutional network [Kipf and Welling 2016]. Gated Graph Sequence Neural Networks (GGNN) incorporates Gated Recurrent Units (GRUs) on top of Graph Neural Networks (GNN), which is a variant of GNN [Li et al. 2015]. Graph Attention Network (GAT) employs a self-attention layer that assigns distinct weights to nodes and their neighboring nodes, thereby facilitating a more accurate representation [Veličković et al. 2017].

3 Motivation

3.1 Smart Contract Vulnerability

This study focuses on two specific vulnerabilities: the reentrant vulnerability and the timestamp dependency vulnerability. In the real world, blockchain networks have sustained significant losses as a result of these two vulnerabilities. For example, the aggregate losses resulting from the Dao incident [Mehar et al. 2019] exceeded \$70 million. The aggregation of publicly available data revealed that the prevalence of smart contracts bearing these two vulnerability labels is considerable, rendering their identification challenging when compared to the relatively limited number of functions impacted by the majority of other contract vulnerabilities.

3.1.1 Reentrant Vulnerability

Solidity offers a more sophisticated syntax for implementing transactions and functions between different addresses as blockchain transactions, with the primary function being the transfer of funds. A reentrant vulnerability arises when the built-in token transfer function (`call.value`) can be invoked recursively from an external call.

Figure 1 illustrates a vulnerable contract from the Bank with a reentrant vulnerability attack, which provides a time deposit function on Ether, enabling users to make deposits that resemble those made in traditional banking institutions. The functions `save()` and `withdrawal()`, which impose a time limit on withdrawals, have been implemented. To address the vulnerability identified in line 12, an attacker may hijack the target contract with the intention of making a malicious withdrawal and thereby stealing the user's entire balance. Once the deposit has been completed, the attack contract is employed to facilitate a withdrawal that aligns with the stipulated conditions, thereby transferring the balance to the attack contract. The transfer transaction will prompt the execution of the fallback function, which has been prepared by the attacker and is, in fact, still the withdrawal code. Since the transfer statement occurs before the state modification, the state can still be evaluated based on the established conditions, and the withdrawal can still occur until the account balance is depleted. Reentrant attacks are contractual vulnerabilities built on the basis of the fallback function, often due to the execution of the state is not instantly updated, resulting in the failure of the judgment conditions caused by the vulnerability of the attack can be attacked.

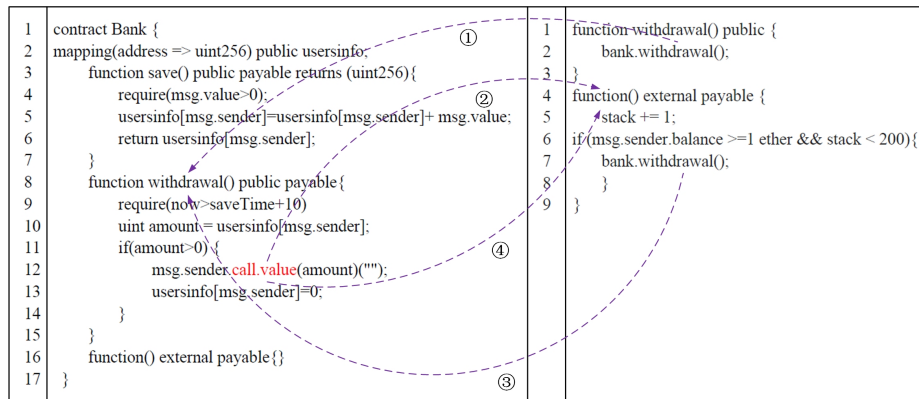


Figure 1: An example of reentrancy vulnerability

3.1.2 Timestamp Dependency Vulnerability

In the context of smart contracts, the `block.timestamp` is utilized as an initial condition for the execution of specific pivotal operations. In the majority of cases, the timestamp is set to the system time of the miner's local computer or server. In the process of mining a block, it is incumbent upon the miner to generate a timestamp for that block. The timestamp of a block may differ from that of other blocks by approximately 900 seconds. Upon receiving a new block that has been validated, the miner verifies that the timestamp of the received block is greater than that of the previously validated block and that the timestamp on the miner's local machine is not more than 900 seconds from that of the received block. The flexibility of the block timestamps set by the miner allows for the manipulation of the outcome of a timestamp dependency smart contract by an adversary or a malicious miner, who may choose different block timestamps. In the event that the contract employs the use of "Now", "StartTime" and "EndTime", which are based on the `block.timestamp`, it is possible for a miner to manipulate the timestamp by a few seconds by modifying the output in a manner that is favorable to them, with the intention of stealing blockchain virtual currency.

A vulnerability in the dependency on timestamps arises when a smart contract employs the value of `block.timestamp` as a component of the invocation condition for a critical operation, such as a transfer of tokens. The value of the `block.timestamp` is generated by the node executing the smart contract (e.g., an anonymous miner), which renders it susceptible to manipulation and attack.

3.2 Observation and Consideration

3.2.1 Fallback Function

As illustrated in Figure 1, the data flow within smart contracts is intricate, and the fallback function call is a pivotal element in the process of contract exploitation.

The fallback function represents a distinctive mechanism inherent to the domain of smart contracts. It is a special function that lacks both a name and parameters. It can be programmed arbitrarily and has no input or return value. In a smart contract, each function is identified by a unique signature. If the signature matches that of the function

of the called contract, the execution proceeds to the corresponding function. In the event that the conditions as mentioned above are not met, the process will instead proceed to the designated fallback function. Additionally, the fallback function is triggered in instances where the transfer is deemed to possess an empty signature. A single fallback function is permitted for each smart contract. The fallback function serves two purposes: (i) when an external user calls a function of a contract that does not exist, the fallback function will be executed automatically; (ii) when an external user transfers Ether to the contract, the fallback function will be executed to receive Ether automatically.

It is of paramount importance to recognize the significance of the fallback function. The existing research methods tend to ignore such a special function with no name and no parameters when constructing graphs for contracts. This has resulted in the focus on reentrant attacks, which highlights the necessity of considering this issue when constructing contract graphs.

3.2.2 Different Program Elements in a Function Are Not Equally Important

Conventional detection methodologies invariably regard all elements within a smart contract function as essentially analogous, thereby precluding the possibility of affording heightened scrutiny to the function that defines the representation of a vulnerability. Indeed, the relative importance of different program elements within a function is not uniform. To illustrate, in a smart contract, the trigger condition for a reentrant vulnerability is invariably accompanied by a call to the `call.value()` function. Similarly, the flow of data during code execution is not equally important. In order to better preserve the features in the graph structure, we propose rules for extracting contract graph nodes and constructing contract edges. The final output of the node information will be more precise and more accurate in detecting relevant vulnerabilities. The specific method will be introduced in detail in the following section.

4 Method

This section delineates the particular methodology employed in the implementation of our model.

Prior to conducting the detection, it is necessary to undertake a series of preprocessing operations on the raw data. The presence of annotations in the code may result in the GNN model processing superfluous information about the smart contract source code, which could potentially introduce noise into the model. Furthermore, the removal of annotations from the source code facilitates a reduction in the input length when feeding into the model. Furthermore, additional blank lines and superfluous spaces were removed in order to reduce the length of the source code further. Incorporating fallback nodes during the construction of the contract graph, as well as edges with fallback functions, facilitates the model's comprehension of the semantic function within the smart contract context, thereby enhancing its accuracy in detecting relevant information. The cleaned source code is then used as the original input for downstream feature extraction.

4.1 Contract Graph Construction and Normalization

The function calls and variable information are extracted from the smart contract source code. The function names and keywords are defined as nodes, and the relationship between variables and functions is converted into edges in the graph.

Previous researchers have consistently emphasized the importance of identifying key nodes and incorporating secondary nodes when constructing contract graphs. However, this approach does not fully capture the actual execution sequence and data flow information inherent to smart contracts. Fallback nodes are incorporated into the flowchart, and three categories of nodes are extracted from the source code: major nodes, minor nodes, and fallback nodes. Preserving the node information allows for more effective utilization of the smart contract’s functionality and enhances the precision of model detection. Referring to previous researchers’ methods for generating edges for smart contract control flow graphs, four types of node types of edges, i.e., control flow, data flow, forward and backward, can be constructed with the addition of backward nodes.

The edges of a contract delineate the process of transferring data flow and control flow, reflecting both the transfer of variable values and the invocation of functions. In this context, the numbering of edges represents the potential sequence of execution for the program, and types are introduced to enhance the differentiation between data flow and control flow. In this work, we define the priority of the edges, which has not been covered in previous research. To illustrate, the dataflow edge type of the account balance node in the reentrant vulnerability graph structure is more effective at identifying contracts containing reentrant vulnerabilities than the sequential flow edge type. The numbering and classification of edges allow for the optimization of semanticized information and the execution logic of the code while preserving the relational structure. The establishment of priorities for edges enables the model to allocate varying levels of attention to different types of edges during the learning process. In the following section, we will present the argument that the accuracy of vulnerability detection can be enhanced by assigning distinct weights to edges with varying priorities through the use of temporally augmented attention networks. The definition and priority setting of edges are illustrated in Table 1.

Syntax	Type	Priority
access{X} assign{X}	Data-flow	1
if{X} if{...}else{X} if{...}then{X} while{X}do{...} for{X}do{...}	Control-flow	2
assert{X} require{X} revert throw		
execute in order of occurrence	Forward	3
fallback function triggers	Fallback	4

Table 1: Summary of edge types

The final generated graph is composed of function nodes, wherein link relations are derived from the original edge links. Function nodes are normalized to comprise the original function features and the variable features associated with them, as illustrated in Figure 2. The function nodes M1, M2, and M3 are associated with the withdraw,

call.value, and deposit functions, respectively. The secondary nodes S1 and S2 are linked to balances[msg.sender] and _amount, respectively. The contract commences at M1 and progresses via the forward edges to attain S1, which subsequently reaches S2 through the data flow and S2 through the control flow to attain M2. S1 then updates itself, thereby triggering the ring data flow edge. It subsequently reaches S2 through the data flow, whereupon S2 triggers the M3 function call through the forward edge.

In light of the limitations of the current smart contract transgraph representation format, it is imperative to streamline the extracted graph, taking into account the potential interference of irrelevant functions, in order to enhance the clarity and precision of the representation. In order to simplify the graph structure, a normalization process is applied to the graph, which is then used to visualize the source code. Additionally, the features of edges between different nodes are labeled.

The contract graph is maintained, retaining all major nodes and eliminating both redundant nodes and edges. The latter are identified through the computation of their embedding vectors, which are then passed on to neighboring major nodes. Edges linked to nodes that have been removed are retained, but their starting or ending nodes are relocated to the corresponding primary nodes. The features of the primary node are updated by aggregating the features of its neighboring removed nodes.

Once processing is complete, the normalized graph is utilized as an input to the graph neural network model. During the training process, the neural network receives a substantial number of normalized graphs constructed from smart contract functions and their corresponding true labels. After a considerable number of training iterations, the neural network generates vulnerability detection results.

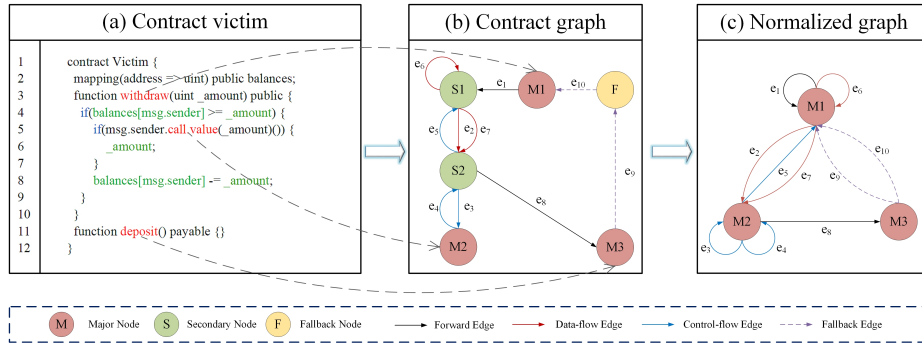


Figure 2: Contract invocation relation extraction method

4.2 Our Model

In this paper, we proposed the ESA model that comprises the following three phases: message propagation, enhanced sequential algorithm, and detection. The general framework of the ESA model is shown in Figure 3.

In the phase of message propagation, the ESA transmits messages in a continuous, chronological sequence along the edges of the graph. Subsequently, the labeling of the entire graph G is computed using a readout function that aggregates the final states of all

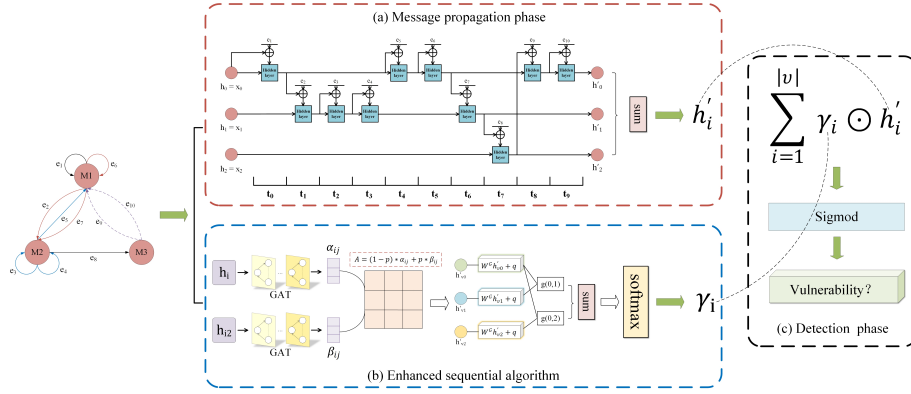


Figure 3: General framework of the ESA model

nodes in G . The readout function is a function of the graph G and is defined as a function of the final state of all nodes in G . Formally, $G = (v, \varepsilon)$, where v consists of all primary nodes and ε contains all edges. $\varepsilon = \{\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n\}$, where ε_k denotes the k_{th} temporal edge.

The message propagation phase entails the transmission of messages along the edges, with each edge being processed in a single time step. At time step 0, the hidden state h_i^0 of each node v_i is initialized using the features of v_i . h_i^0 is the embedding of node v_i in the initial state (state 0), and the hidden state of the node is updated progressively during the message propagation. At time step n , the message traverses the n_{th} time edge ε_n and updates the hidden state of the end node. In particular, the message m_n is initially calculated based on the hidden state h_n^{start} of the initial node of ε_n and the edge type t_n :

$$x_n = h_n^{start} \oplus t_n \quad (1)$$

$$m_n = W_n x_n + q_n \quad (2)$$

Where \oplus represents the splice symbols, and the matrix W_n and the bias vector q are the network parameters. The original message x_n contains information from ε_n and the starting node of the edge ε_n itself, which is then converted into a vector embedding using W_n and q . After receiving the information features, the end node of the edge can update its own hidden layer with the input information as well as the history hidden layer. h_{ε_n} is updated according to the following:

$$\hat{h}_{\varepsilon_n} = \tanh(Um_n + Zh_{\varepsilon_n} + q_1) \quad (3)$$

$$h'_{\varepsilon_n} = \text{softmax}(R\hat{h}_{\varepsilon_n} + q_2) \quad (4)$$

Where U , Z , and R are matrices and q_1 and q_2 are bias vectors. \tanh function maps the transformed feature representation to the range between -1 and 1.

Attention mechanism module:

This module employs the graph attention mechanism to update node and edge features. Additionally, it extracts the attention coefficients from the updated control flow graph, which are then used to generate graph features. The combination of attention

with the processes above allows for a comprehensive consideration of the relationship between nodes and the node's inherent characteristics, thereby facilitating the extraction of graph features with enhanced characterization abilities and the fulfillment of the timing enhancement objective.

The calculation of candidate hidden states is performed using the graph attention mechanism, which is employed to determine the attention score between nodes. The calculation of the attention score can be expressed by equation (5):

$$e_{ij} = \text{LeakyReLU}(o[W h_i \parallel W h_j]) \quad (5)$$

Where h denotes the feature vector of the node and o is the learnable attention weight vector, $o \in R^{2F'}$. Let h_n denote the node features in the graph after GNN training update, where n is the number of nodes. h_i and h_j are the feature vectors of the i_{th} and j_{th} nodes. \parallel denotes the tandem operation of the vectors. W is a learnable parameter matrix, $W \in R^{F' \times F}$, Where F denotes the dimension of the node embedding and F' denotes the dimension of the updated node embedding.

Attention weights [Veličković et al. 2017] are calculated as shown in equation (6). For a given node v_i , the attention scores between itself and its neighbor v_j are computed by *softmax* operation to generate the attention weights α_{ij} . These weights indicate the importance of the influence of node v_i on its neighbor node v_j .

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N(i)} \exp(e_{ik})} \quad (6)$$

Where $\exp(x)$ denotes the value of the natural exponential function e^x , and *softmax* is a commonly used activation function.

Similarly, the attention coefficients β_{ij} that characterize the edges are obtained by individually updating the attention to the features of the edges.

$$\varepsilon_{ij} = \text{LeakyReLU}(o'[W' h_{i2} \parallel W' h_{j2}]) \quad (7)$$

$$\beta_{ij} = \text{softmax}_j(\varepsilon_{ij}) = \frac{\exp(\varepsilon_{ij})}{\sum_{k \in N(i)} \exp(\varepsilon_{ik})} \quad (8)$$

The attention coefficient matrices α_{ij} and β_{ij} for all node features and edge features are obtained by the above formulae. The former is learned from all node features, while the latter is computed separately from h_{i2} , expressing edge features in node feature h_i . The new attention coefficient matrix A is obtained by summing the weights, which enhances the influence of the temporal features of the edges on the classification results.

$$A = (1 - p) * \alpha_{ij} + p * \beta_{ij} \quad (9)$$

After adding the attention coefficients obtained from the computation of the features of the edges, the new attention coefficients A are used to aggregate the features of the neighboring nodes and update the node information.

$$h'_{vi} = \sigma\left(\sum_{j \in N(i)} A_{ij} W'' h_j + h_i\right) \quad (10)$$

Where σ denotes the *sigmoid* activation function and W'' is the learnable parameter matrix. After we get the updated node information h'_{vi} , we use the self-attention

mechanism to transform it linearly. The procedure is shown in equation (11):

$$g_i = \tanh(W^G h'_{vi} + q) \quad (11)$$

Where W^G is the weight matrix, and q is the bias vector. \tanh function maps the transformed feature representation to a range between -1 and 1, thus generating the potential weights of the normalized graph.

In the self-attention mechanism, we further integrate node features and edge features into the global representation of the graph. Self-attention effectively captures the dependencies and semantic information between nodes and edges. In this mechanism, we perform a weighted summation of the fusion of all node features and edge features to obtain the contribution of nodes to the overall graph features:

$$\gamma_i = \text{softmax}(g_i) = \frac{\exp(g_i)}{\sum_{k \in N(i)} \exp(g_k)} \quad (12)$$

The *softmax* function is employed to ascertain the proportion of influence, which is often referred to as weight, exerted by node g_i relative to its neighboring nodes g_k . This value is constrained within the range of 0 to 1.

The self-attention mechanism is utilized to discern potential interconnections between the class features of subgraphs. This enables the aggregation of features pertaining to each node and the interconnections between them, thereby facilitating the acquisition of a comprehensive representation of the graph. This representation encapsulates the collective features of the graph. This attentional mechanism facilitates a more profound comprehension of the structural and semantic information inherent to the graph while simultaneously integrating global insights pertaining to nodes and edges, thereby enhancing the model's capacity to represent and generalize graph properties.

Detection phase: The ESA computes the label \hat{y} of G by successively traversing all edges in G and reading out the final hidden states of all nodes:

$$\hat{y} = \text{sigmoid} \sum_{i=1}^{|v|} \gamma_i \odot h'_i \quad (13)$$

The \odot symbol represents the element-by-element product, while the *sigmoid* function maps the result to either 0 or 1.

During the training process, the network receives a substantial number of normalized graphs, which have been constructed from smart contract functions and their corresponding true labels. Subsequently, the trained model is employed to absorb the normalized graph and generate vulnerability detection labels, with the entire process being fully automated.

4.3 Algorithm

It is worth noting that, in order to optimize the utilization of the connection information between nodes, we proposed an enhanced sequential algorithm for node feature aggregation. This algorithm considers not only the importance of different nodes but also the priority of the edges between nodes as a reference. The detailed process is illustrated in Algorithm 1.

First, the node features h_i are used as inputs to initialize the learnable attention weight vectors and parameter matrices (lines 1-2). where the training parameters W are shared in

Algorithm 1: Enhanced sequential node feature aggregation algorithm

Input: h : Node features, v : Nodes in the graph.
Output: γ_i : Figure-level attention coefficient.

- 1 Initialize learning parameter matrix W, W', W'' ;
- 2 Initialize learnable attention weight vectors o, o' ;
- 3 **for each node** $v_i \in v$ **do**
- 4 **for each node** $v_j \in N_{v_i}$ **do**
- 5 Update attention score calculation e_{ij} by equation (5);
- 6 Update attention score calculation ε_{ij} by equation (7);
- 7 **end**
- 8 **for each node** $v_j \in N_{v_i}$ **do**
- 9 Update attention score calculation α_{ij} by equation (6);
- 10 Update attention score calculation β_{ij} by equation (8);
- 11 **end**
- 12 Update attention coefficient matrix A by equation (9);
- 13 Update node features h'_{v_i} by equation (10);
- 14 **end**
- 15 Initialize parameter matrix W^G, q ;
- 16 **for each node** $v_i \in v$ **do**
- 17 **for each node** $v_j \in N_{v_i}$ **do**
- 18 Calculate attention score g_i by equation (11);
- 19 **end**
- 20 **for each node** $v_j \in N_{v_i}$ **do**
- 21 Calculate figure-level attention coefficient γ_i by equation (12);
- 22 **end**
- 23 **end**
- 24 return γ_i ;

equation (5) and equation (7), and the training parameters W' are shared in equation (6) and equation (8). The loop traverses all nodes v (lines 3-14). In this loop, four operations are performed on node v_i : the attention scores of all node features and edge features are computed (lines 4-7); the normalized attention weights are computed (lines 8-11); the new attention coefficients, A , obtained after adding the features of the edges are computed (line 12); and the node features h'_{v_i} are updated. (Line 13). Subsequently, the steps mentioned above facilitate the aggregation of global information pertaining to the node features, subsequent to the input of said features and the adjacency matrix. Subsequently, the algorithm proceeds to the second phase. The parameter matrices W^G and q are initialized (line 15), and then a loop is executed through all nodes v (lines 16-23). During this loop, the attention weights g_i are computed (lines 17-19), which are then normalized to yield the graph-level attention coefficients γ_i (lines 20-22).

We augment the temporal attributes of contract graph edges through the implementation of the graph attention mechanism, subsequently integrating it with the self-attention mechanism to derive the attention coefficient of the final features aggregated at the nodes. This coefficient serves to quantify the nodes' contribution to the overall graph features.

The algorithmic approach allows for a more profound comprehension of the structural and semantic information inherent to the graph. The integration of global insights into node features enables the model to preserve the original data features with greater fidelity and enhance the accuracy of detection.

5 Experiments

In order to evaluate the performance of our method, we designed the following research questions:

RQ1: What are the pivotal parameters in the model, and how can they be identified in order to optimize the model's performance at its data values?

RQ2: To what extent is our method effective in detecting smart contract vulnerabilities in comparison to cutting-edge vulnerability detection methods?

RQ3: With the introduction of more features, can it help the model to improve the results of vulnerability detection?

RQ4: Does our research component face threats to its validity?

Subsequently, the experimental setup will be presented, after which the research mentioned above questions will be addressed in turn.

5.1 Experimental Settings

5.1.1 Dataset

The dataset was obtained from the open dataset of [Zhuang et al. 2021], which comprises over 40,000 real-world Ethereum smart contracts involving large-scale transactions. The authors have made their dataset publicly available for validating the effectiveness of their system in handling vulnerabilities. The underlying real-world labels of the contract functions have been provided by experts. The publicly available dataset utilized in this paper can be accessed via the following URL: <https://github.com/Messi-Q/Smart-Contract-Dataset>.

The initial step involves the importation of the dataset into the Smartbug tool, which contains nine detection tools. Subsequently, the mainstream static analysis tools are employed to conduct a preliminary scanning of all contract functions. This is followed by the automatic labeling of the functions according to the vulnerability reports output by the tools. For the samples with high confidence in the automatic labeling results, three security experts are organized to cross-validate the labels. Concurrently, 5% of the automatically labeled samples are randomly selected and manually verified to assess the accuracy of the automatic labeling. The automated tool is responsible for generating approximately 60% of the labels, while the remaining 40% are subjected to manual double-checking and confirmation. The final dataset contains 13,476 smart contracts, of which 5,683 are labeled with reentrant vulnerabilities, 5,037 are labeled with timestamp dependency vulnerabilities, and 2,756 no-vulnerability labels.

5.1.2 Experimental Environment

We have built a deep learning framework based on PyTorch to validate the performance of the ESA model. The language environment Python version is 3.8, and the experiments follow the protocols and libraries needed for deep learning, more detailed experimental environment is shown by Table 2.

Experimental environment details	
Operating system	Microsoft Windows 11
Processor	13th Gen Intel(R)Core(TM) i7-13700H
Architecture	64-Bit
GPU	NVIDIA GeForce RTX 4060
Memory	16GB
Language	Python
Framework	PyTorch 1.10.2
Libraries used	keras, sklearn, docopt
IDE	PyCharm

Table 2: Details of experimental environment parameters

5.1.3 Evaluation Indicators

In order to conduct a comprehensive and objective evaluation of the performance of our model, we have selected accuracy, recall, precision, and F1-score as our evaluation metrics. Among the metrics as mentioned earlier, accuracy gauges the overall accuracy of the model’s classification. Recall assesses the model’s capacity to identify positive samples, which is well-suited to scenarios emphasizing omission or underreporting. Precision evaluates the model’s precision in predicting positive samples, which is well-suited to scenarios emphasizing misclassification. F1-score is well-suited to scenarios where there is a trade-off between precision and recall, and can be used to evaluate the overall performance of the model. The formulas for each metric are provided below:

$$Accuracy = \frac{true\ positive + true\ negative}{Number\ of\ total\ samples} \quad (14)$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \quad (15)$$

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \quad (16)$$

$$F1 - score = 2 \times \frac{Recall \times Precision}{Recall + Precision} \quad (17)$$

5.1.4 Validation Methods

The adam optimizer was employed in the ESA model [Kingma 2014]. We found out the optimal settings of the parameters through continuous experiments. Most models are too complex, resulting in noise in the training data, and a problem that often arises is overfitting. Overfitting can be attributed to two primary causes: an excess of feature dimensions (or parameters) or an insufficient quantity of data. This phenomenon manifests as a fitted function that accurately passes through the training set but exhibits suboptimal predictive performance on new data. For smaller datasets, EDA [Wei and Zou 2019] techniques can be used to improve the performance of text categorization tasks.

To overcome the overfitting problem, we chose to use HoldOut cross-validation. For each dataset, we randomly select 80% of it as the training set and the other 20% as the test set, perform multiple tests and report the average results.

5.2 Experiment Analysis

5.2.1 Answer RQ1: The Impact of Various Parameters on Model Performance

Since the performance of deep neural networks is usually affected by their hyperparameters, in this experiment, we consider three pivotal parameters: epoch, learning rate and patience, which are used to investigate whether the parameter settings are favorable for the vulnerability detection task.

Epoch, learning rate, and patience are particularly important parameters in machine learning tasks, and they have a huge impact on the predictive effectiveness of the model. The learning rate facilitates the network's convergence, thereby influencing the detection performance. An excessive learning rate may result in oscillatory behavior or even failure to converge. Conversely, an insufficient learning rate may lead to slow convergence or even gradient disappearance, which in turn may fail to converge. One epoch represents the entirety of the data fed into the network for training purposes. The size of the epoch significantly impacts the model's fitting ability. Patience, in this context, refers to the degree of tolerance for performance changes during the training process. When the model's performance does not demonstrate a notable improvement within a specified number of iterations, it is essential to exercise patience and allow the training to continue, rather than prematurely concluding it and potentially missing further improvements.

We chose to detect the dataset containing the reentrant vulnerability label to reveal the above effects, and the performance of ESA with different parameter settings is shown in Figure 4.

In evaluating the influence of the epoch on the model's performance, the convergence of both accuracy and loss is a crucial consideration. As illustrated in Figure 4 (a), the accuracy of all four vulnerability types exhibits a marked increase as the number of epochs rises, reaching a peak at data point 50. Subsequently, the detection performance reaches a steady state, indicating that 50 epochs are sufficient for training the neural network. Based on the loss curve in Figure 4 (b), it can be seen that in the first 50 training rounds, the loss value shows a pronounced decreasing tendency, and the model learning effect is pronounced; in the 50-150 training rounds, the decreasing tendency slows down and tends to converge; after 150 training rounds, the model tends to converge. Taking this into account, we choose epoch 170, when the model has the highest accuracy. We can also see that our model still shows stability even if the number of training rounds is getting higher and higher. The ESA model has superior convergence speed and robustness. Figure 4 (c) illustrates the effect of different learning rates. In this experiment, we start with a small learning rate and then scale up the value to find a suitable learning rate. In terms of performance, we choose 0.001 as the learning rate for our model. Figure 4 (d) illustrates the impact of varying patience levels. The model exhibits optimal performance at a patience level of 200.

5.2.2 Answer RQ2: Performance Comparison

The objective of the RQ1 study is to examine the efficacy of our methodology in identifying smart contract vulnerabilities within the same dataset, in comparison to existing methods that represent the current state of the art. In order to address this question, we undertake a comparative analysis of our approach with a range of cutting-edge smart contract vulnerability detection efforts, including:

Oyente [Luu et al. 2016]: a well-known symbolic verification tool for smart contract vulnerability detection that performs symbolic execution on CFGs to check for

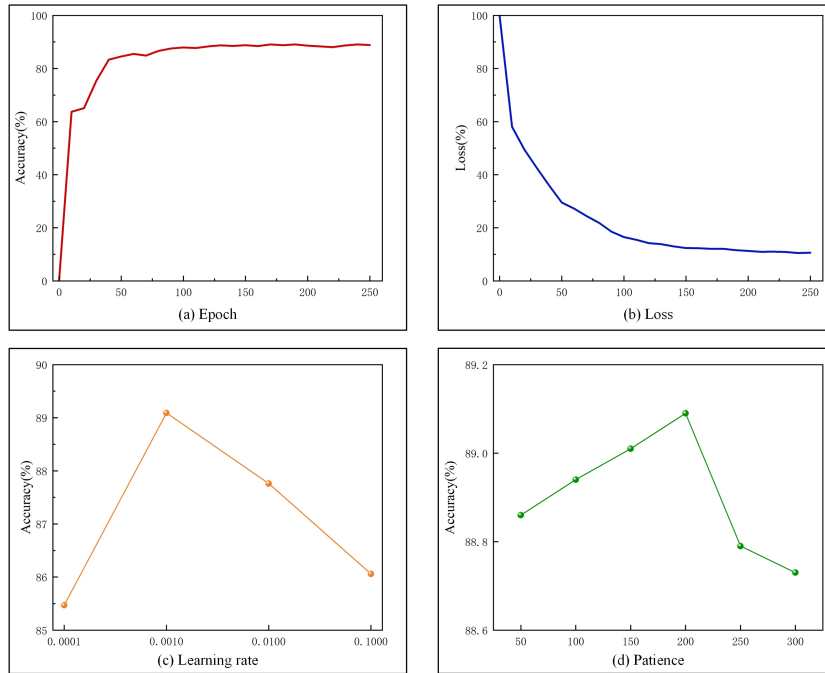


Figure 4: Impact of pivotal parameters in ESA. (a) Impact of epoch on accuracy (b) Impact of epoch on loss (c) Impact of learning rate on accuracy (d) Impact of patience on accuracy

vulnerability patterns. Mythril [Mueller 2017]: a security analysis methodology that uses concatenation analysis, taint analysis, and control flow checking to detect smart contract vulnerabilities. Smartcheck [Tikhomirov et al. 2018]: an extensible static analysis tool for discovering smart contract code vulnerabilities. securify [Tsankov et al. 2018]: a formal verification-based vulnerability detection tool for Ethereum smart contracts that checks for compliance and violation patterns. Slither [Feist et al. 2019]: a static analysis framework designed to discover smart contracts by converting them into an intermediate representation of SlithIR problems in Ethereum smart contracts. DR-GCN [Zhuang et al. 2021]: a contract graph approach to detecting smart contract vulnerabilities, which converts smart contract source code into a contract graph structure with a high semantic representation and builds a security vulnerability detection model using a graph convolutional neural network. TMP [Zhuang et al. 2021]: a deep-learning source-code level vulnerability detection tool based on graph representation and graph neural networks. It converts contracts into normalized graphs and then uses a deep learning model called Temporal Message Propagation Network (TMP) for vulnerability detection. Peculiar [Wu et al. 2021]: a model for detecting smart contract vulnerabilities based on key data flow graphs using a pre-training technique focusing on key data flow graphs that are not too complex. AME [Liu et al. 2021]: The authors combine deep learning with expert models to develop automated tools to extract expert patterns from source code and then

convert the code into semantic graphs to extract deep graph features.

Methods	Reentrancy				Timestamp dependency			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
Smartcheck	53.81	24.21	35.36	26.09	46.03	58.30	43.53	48.96
Mythril	62.41	73.60	41.22	52.85	61.74	45.76	53.82	49.43
Oyente	63.35	58.87	42.36	49.26	63.87	48.21	53.10	50.50
Securify	72.39	64.83	59.63	61.99	—	—	—	—
Slither	75.57	73.89	71.43	72.60	71.36	69.78	68.26	68.96
DR-GCN	77.34	77.04	73.42	75.11	77.30	78.23	73.11	75.57
Perculiar	79.68	78.23	77.63	77.93	—	—	—	—
TMP	80.47	78.97	75.05	76.89	81.15	79.96	76.87	78.28
AME	85.63	84.07	82.94	83.49	84.39	83.25	81.75	82.47
ESA	89.09	88.37	84.56	86.42	88.49	87.05	88.55	87.80

Table 3: Comparison of experimental results for different models

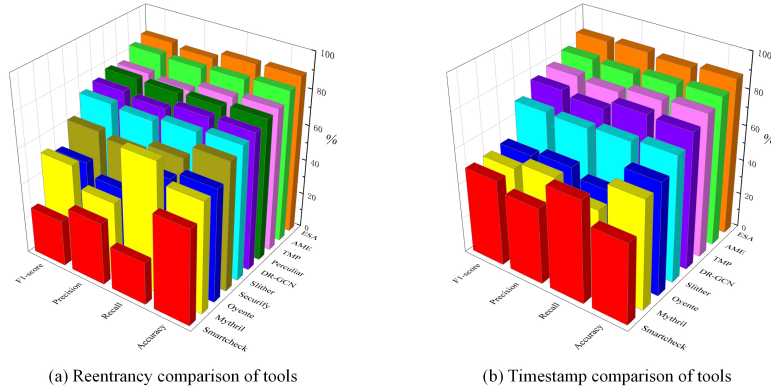


Figure 5: Visual representation of the results of the comparison experiment

Figure 5 (a) and Figure 5 (b) show the comparative results of reentrant vulnerability and timestamp dependency vulnerability detection, with the ten rows from front to back denoting the Smartcheck, Mythril, Oyente, Securify, Slither, DR-GCN, Perculiar, TMP, AME, and ESA methods, respectively. For each row in the figure, accuracy, recall, precision, and F1-score are shown from left to right.

Reentrant vulnerability detection comparison:

A comparison is presented between our method and nine existing methods on the task of reentrant vulnerability detection. The performance of the various methods is presented in Table 3, which includes metrics such as accuracy, recall, precision, and F1-score. Figure 5 (a) provides a visual representation of the reentrant vulnerability detection results. From the detection results, the following conclusions can be drawn. Firstly, it is evident that traditional non-deep learning methods have not yet achieved a satisfactory accuracy rate on the reentrant vulnerability detection task. For example, the cutting-edge traditional method (i.e., Slither) has an accuracy rate of 75.57%. Secondly, the proposed

method demonstrates superior performance compared to existing methods in the domain of reentrant vulnerability detection. In particular, the model demonstrates an accuracy of 89.09%, which represents a 13.52% improvement over traditional methods. Even the cutting-edge deep learning method (i.e., AME) has an accuracy of only 85.63%, which is 3.46% lower than our method. These improvements may be attributed to our incorporation of key variables and rich dependencies between program elements in smart contracts, along with the assignment of greater weight to edges that may contain reentrant vulnerabilities, thereby maximizing the utilization of semanticized information and the execution logic of the code.

Timestamp dependency vulnerability detection comparison:

In the timestamp dependency vulnerability detection task, we compare seven other methods with our approach. The comparative results are presented in Figure 5 (b). Furthermore, in accordance with the findings of the reentrant vulnerability detection analysis, our model demonstrates the most optimal performance across all four metrics. The accuracy of our model is 89.49%, which is 18.13% higher than the cutting-edge traditional method (i.e., Slither) and 4.10% higher than the cutting-edge deep learning method (i.e., AME). The prevailing methodologies for detecting timestamp dependency vulnerabilities are predicated on the presence of a `block.timestamp` statement within a function. However, these approaches demonstrate a paucity of consideration for the surrounding control flow, data flow, and contract state update order.

By examining existing methods, it can be posited that the reason for the low precision and recall of traditional methods is that they rely heavily on simple and fixed patterns to detect vulnerabilities. For instance, Mythril necessitates the utilization of sophisticated methodologies, such as taint analysis or manual auditing, to discern reentrant vulnerabilities. This is achieved by solely examining whether there are no internal function calls subsequent to the `call.value` call. Conversely, Slither and Smartcheck rely on predefined code patterns to identify reentrant vulnerabilities. However, these patterns may prove insufficiently comprehensive or adaptable to encompass the full spectrum of potential vulnerability scenarios. In order to detect reentrant vulnerabilities, both Slither and Smartcheck initially identify external calls within a function and then examine the occurrence of state change operations subsequent to the external call. However, due to the complexity of external calls in smart contracts, these traditional pattern-based detection methods may prove inadequate in covering all potential scenarios.

Compared to deep learning-based detection methods, our approach still outperforms them. The main reason is that all these methods lose some of the key information within the smart contract code during code representation learning. For example, in Peculiar, the code graph is constructed based on data flow information, and each node in its graph represents only one variable within the code. As a result, this approach can only partially capture features related to changes in state variables within the code. However, the lack of interaction operations and relationships between interactions and state variables in the code graph, a data-flow-related code graph, makes it difficult for Peculiar to learn all the necessary functionality from this one-sided representation of the code. Similar to the limitations of static analysis tools, TMP also uses pre-defined rules to identify nodes in CFGs where external calls exist, which may not cover all possible scenarios and may incorrectly remove externally called nodes from CFGs. Meanwhile, in the process of simplifying the CFG, the TMP retains only those nodes in the CFG that contain external calls or state variables and merges these nodes to normalize the CFG. While node merging reduces the size of the graph, it also removes control flow information between statements, which is critical for reentrant vulnerability detection. Therefore, this oversimplified graph may limit the ability of TMP to detect reentrant vulnerabilities. The

accuracy of DRGCN is lower than that of TMP because DR-GCN is unable to capture temporal information induced by data flow and control flow, which is explicitly taken into account by the use of ordered edges in TMP. In addition, traditional graph neural networks (e.g., GCN) fail to emphasize the different importance of different nodes; our proposed ESA not only takes into account the importance of different nodes but also the importance of different types of edges, and this optimization better improves the accuracy of our model in detecting vulnerabilities. Although AME combines local expert patterns with global graph features, its proposed expert pattern features have great limitations and these rules do not cover all smart contracts.

5.2.3 Answer RQ3: Ablation Experiment

In order to investigate whether the incorporation of the fallback feature in contract graph construction and our proposed enhanced sequential algorithm are effective in smart contract vulnerability detection, we conducted ablation experiments to examine whether the two features proposed above can improve the vulnerability detection results by comparing the vulnerability detection results.

Methods	Reentrancy				Timestamp dependency			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
ESA-F0	79.04	50.53	80.82	62.18	79.41	87.18	76.89	81.71
ESA-F1	82.23	75.95	73.61	74.76	83.57	80.51	82.78	81.63
ESA-F2	85.43	79.33	77.16	78.23	84.64	77.72	85.46	81.41
ESA	89.09	88.37	84.56	86.42	88.49	87.05	88.55	87.80

Table 4: Comparison of ablation experiment results

As shown in Table 4, where ESA-F0 represents that the backoff function is not considered in the construction of the contract graph, while the timing enhanced algorithm module is removed; ESA-F1 is the model that does not consider the backoff function of the smart contract, but the timing enhanced module is set up; and ESA-F2 is the model that adds the backoff function of the smart contract to the construction of the contract flowchart but does not include the timing enhanced module.

The ablation experiments show that adding the fallback function and the enhanced sequential algorithm to the contract flowchart construction is effective for smart contract vulnerability detection. The combination of the two can maximize the detection performance and shows a large advantage in terms of accuracy, recall, precision and F1-score.

5.2.4 Answer RQ4: Threats to Validity

External Threats: The primary external threat to the veracity of our research findings is the reliability of the dataset. The publicly available dataset provided by [Zhuang et al. 2021] compiles smart contracts written in Solidity and labeled from a variety of sources. The dataset comprises two categories. The first category comprises contracts from other public works on github, which other researchers manually labeled. However, the process of manual labeling is inherently labor and expertise intensive and is therefore susceptible to subjectivity. The second category of datasets is constructed manually through the injection of vulnerabilities. The code logic of these injections is relatively straightforward

and may only involve some evident vulnerability patterns. Vulnerability injection-based methods generate samples through templates and are therefore unable to fully cover all forms of a given vulnerability type. Furthermore, the injected vulnerability code is isolated from the original smart contract code context and may not accurately reflect the actual smart contract scenario. The dataset employed in this study encompasses only two categories of vulnerabilities, accompanied by corresponding transformation criteria. This may restrict the scope of vulnerability detection to the categories mentioned above, potentially hindering the identification of other forms that are not represented in the dataset.

Internal Threats: There are some internal threats in our implementation process. Firstly, there are numerous uncertainties present within the data processing and model training stages, including issues such as sample imbalance and hyperparameter tuning. These potential issues may impact the efficacy of our proposed methodology. As the size of the dataset increases, the number of variables that require control in the experiment also increases, which can result in unexpected bias in the experimental results. To mitigate this potential threat, the same proportion randomly disrupts the dataset, and the validation set is used to test the generalization performance of the model. Secondly, the syntactic features of the two vulnerabilities are summarised, and the construction criteria for nodes and edges are defined manually based on the control and data flows in the source code. As these criteria are defined subjectively, they may not be entirely accurate and may fail to account for some exceptional cases. For instance, the introduction of new syntactic features by a Solidity language update presents a challenge in enumerating all potential construction criteria. Consequently, an avenue for enhancing the existing methodology would be to discern increasingly detailed construction regulations for nodes and edges that encapsulate a greater range of vulnerability-related semantics and exclude superfluous code components during the training process.

Construct Threats: In this paper, construct validity threats are mainly related to biases in the choice of benchmarks and evaluation functions. In our methodology, we transform the source code of the smart contract into a graph representation of the construct. Although control flow graphs are the most prevalent code representation, alternative representations, such as value flow graphs or program dependency graphs, are capable of reflecting a subset of the functionality inherent to smart contract code. Furthermore, four performance metrics—accuracy, recall, precision, and F1-score—are employed to assess the efficacy of vulnerability detection. While these four metrics have been widely used in previous research, other metrics are sometimes employed, such as G-mean. G-mean is a particularly useful metric as it reflects the performance of the classifier in dealing with unbalanced datasets more accurately than simple classification accuracy. In the event of a considerable discrepancy between the number of instances of the positive and negative classes within a given dataset, the classifier may exhibit a degree of bias. In such instances, the G-mean may prove a superior metric for reflecting this bias.

Conclusion Threats: The conclusion is that threats to our approach are mainly related to the ability to generalize the dataset. The dataset was constructed from 9,000 smart contracts written in the Solidity language, covering versions 0.5 to 0.8. While our model training incorporates newer smart contracts, the selected samples may still be imperfect and may lack representation of some extreme cases. Moreover, the ecosystem of Solidity smart contracts is undergoing rapid evolution, with frequent updates to the syntax. Our approach is contingent upon the established conventions for constructing contract graphs, which may prove inadequate for detecting vulnerabilities that employ more recent semantics.

6 Conclusion and Future Work

This paper proposes a vulnerability detection model based on an enhanced sequential algorithm for the automatic detection of reentrant and timestamp dependency vulnerabilities in smart contracts. The model eliminates extraneous data from the source code of smart contracts, such as comments, blank lines, and excessive spacing that do not contribute to the functionality or structure of the code. Additionally, the model incorporates the smart contract fallback function into the construction of the contract graph, ensuring the inclusion of essential information while reducing the complexity of the graph. The enhanced sequential algorithm of attention not only considers the importance of different nodes but also takes the importance of different types of edges into account, thus enabling the model to fully preserve the characteristics of the original data. In comparison to traditional methodologies and existing deep learning techniques, our model demonstrates superior performance in the detection of both reentrant vulnerabilities and timestamp dependency vulnerabilities.

In future work, three avenues of learning will be pursued. Firstly, the present study focused on the detection of only two types of vulnerabilities. It would be beneficial for future research to consider a greater number of types of vulnerabilities in order to enhance the overall performance of vulnerability detection. Secondly, this paper assesses methodologies for smart contracts written in Solidity. In the future, we intend to develop comprehensive graph neural network detection techniques for diverse programming languages utilized in smart contracts. Thirdly, vulnerability detection is conducted within functions. Subsequent research will extend this approach to encompass cross-functional vulnerability detection.

References

- [Brent et al. 2018] Brent, L.; Jurisevic, A.; Kong, M.; Liu, E.; Gauthier, F.; Gramoli, V.; Holz, R.; Scholz, B. Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981, 2018.
- [Cai et al. 2023] Cai, J.; Li, B.; Zhang, J.; Sun, X.; Chen, B. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *Journal of Systems and Software*, 2023, 195, 111550.
- [Fan et al. 2019] Fan, W.; Ma, Y.; Li, Q.; He, Y.; Zhao, E.; Tang, J.; Yin, D. Graph neural networks for social recommendation. In *Proceedings of the The world wide web conference*, 2019, pp. 417–426.
- [Feist et al. 2019] Feist, J.; Grieco, G.; Groce, A. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
- [Goller and Kuchler 1996] Goller, C.; Kuchler, A. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the Proceedings of international conference on neural networks (ICNN'96)*. IEEE, 1996, Vol. 1, pp. 347–352.
- [Grishchenko et al. 2018] Grishchenko, I.; Maffei, M.; Schneidewind, C. A semantic framework for the security analysis of ethereum smart contracts. In *Proceedings of the Principles of Security and Trust: 7th International Conference, POST 2018, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer, 2018, pp. 243–269.
- [He et al. 2019] He, J.; Balunović, M.; Ambroladze, N.; Tsankov, P.; Vechev, M. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 531–548.

- [He et al. 2023] He, L.; Zhao, X.; Wang, Y.; Yang, J.; Sun, X. GraphSA: Smart Contract Vulnerability Detection Combining Graph Neural Networks and Static Analysis. In *ECAI 2023*; IOS Press, 2023; pp. 1020–1027.
- [Hildenbrandt et al. 2018] Hildenbrandt, E.; Saxena, M.; Rodrigues, N.; Zhu, X.; Daian, P.; Guth, D.; Moore, B.; Park, D.; Zhang, Y.; Stefanescu, A.; et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *Proceedings of the 2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.
- [Jiang et al. 2018] Jiang, B.; Liu, Y.; Chan, W.K. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.
- [Kalra et al. 2018] Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. Zeus: analyzing safety of smart contracts. In *Proceedings of the Ndss*, 2018, pp. 1–12.
- [Kipf and Welling 2016] Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [Kingma 2014] Kingma, D.P. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [Li et al. 2015] Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.
- [Liu et al. 2018] Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; Roscoe, B. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, 2018, pp. 65–68.
- [Liu et al. 2021] Liu, Z.; Qian, P.; Wang, X.; Zhu, L.; He, Q.; Ji, S. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint arXiv:2106.09282*, 2021.
- [Luu et al. 2016] Luu, L.; Chu, D.H.; Olickel, H.; Saxena, P.; Hobor, A. Making smart contracts smarter. In *Proceedings of the Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
- [Mehar et al. 2019] Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. *Journal of Cases on Information Technology (JCIT)* 2019, 21, 19–32.
- [Mueller 2017] Mueller, B. A framework for bug hunting on the ethereum blockchain. *ConsenSys/mythril*, 2017. <https://github.com/ConsenSys/mythril>.
- [Mou et al. 2016] Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Proceedings of the AAAI conference on artificial intelligence*, 2016, Vol. 30.
- [Phan et al. 2017] Phan, A.V.; Le Nguyen, M.; Bui, L.T. Convolutional neural networks over control flow graphs for software defect prediction. In *Proceedings of the 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2017, pp. 45–52.
- [Scarselli et al. 2008] Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The graph neural network model. *IEEE transactions on neural networks*, 2008, 20, 61–80.
- [Sak et al. 2014] Sak, H.; Senior, A.W.; Beaufays, F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling, 2014.
- [Suneja et al. 2020] Suneja, S.; Zheng, Y.; Zhuang, Y.; Laredo, J.; Morari, A. Learning to map source code to software vulnerability using code-as-a-graph. *arXiv preprint arXiv:2006.08614*, 2020.
- [Tann et al. 2018] Tann, W.J.W.; Han, X.J.; Gupta, S.S.; Ong, Y.S. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632*, 2018.

- [Tikhomirov et al. 2018] Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. Smartcheck: Static analysis of ethereum smart contracts. In Proceedings of the Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, 2018, pp. 9–16.
- [Torres et al. 2019] Torres, C.F.; Steichen, M.; et al. The art of the scam: Demystifying honeypots in ethereum smart contracts. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1591–1607.
- [Tsankov et al. 2018] Tsankov, P.; Dan, A.; Drachler-Cohen, D.; Gervais, A.; Buenzli, F.; Vechev, M. Securify: Practical security analysis of smart contracts. In Proceedings of the Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 2018, pp. 67–82.
- [Veličković et al. 2017] Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; Bengio, Y. Graph attention networks. arXiv preprint arXiv:1710.10903, 2017.
- [Wang et al. 2021] Wang, W.; Song, J.; Xu, G.; Li, Y.; Wang, H.; Su, C. Blockchain-Enabled Authentication Handover With Efficient Privacy Protection in SDN-Based 5G Networks. IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING, 2021, 8, 1133–1144.
- [Wei and Zou 2019] Wei, J.; Zou, K. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. arXiv preprint arXiv:1901.11196, 2019.
- [Wu et al. 2021] Wu, H.; Zhang, Z.; Wang, S.; Lei, Y.; Lin, B.; Qin, Y.; Zhang, H.; Mao, X. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2021, pp. 378–389.
- [Yamaguchi et al. 2014] Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In Proceedings of the 2014 IEEE symposium on security and privacy. IEEE, 2014, pp. 590–604.
- [Yu et al. 2021] Yu, X.; Zhao, H.; Hou, B.; Ying, Z.; Wu, B. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN). IEEE, 2021, pp. 1–8.
- [Zhen et al. 2024] Zhen, Z.; Zhao, X.; Zhang, J.; Wang, Y.; Chen, H. DA-GNN: A smart contract vulnerability detection method based on Dual Attention Graph Neural Network. Computer Networks, 2024, 242, 110238.
- [Zhang et al. 2019] Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 783–794.
- [Zhang and Liu 2022] Zhang, Y.; Liu, D. Toward vulnerability detection for ethereum smart contracts using graph-matching network. Future Internet, 2022, 14, 326.
- [Zhao et al. 2021] Zhao, L.; Li, Z.; Al-Dubai, A.Y.; Min, G.; Li, J.; Hawbani, A.; Zomaya, A.Y. A novel prediction-based temporal graph routing algorithm for software-defined vehicular networks. IEEE Transactions on Intelligent Transportation Systems, 2021, 23, 13275–13290.
- [Zhuang et al. 2021] Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart contract vulnerability detection using graph neural networks. In Proceedings of the Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, 2021, pp. 3283–3290.