International Journal on Semantic Web and Information Systems
Volume 20 • Issue 1

Particle Swarm Algorithm for Smart Contract Vulnerability Detection Based on Semantic Web

Tao Feng, School of Computer and Communication, Lanzhou University of Technology, China
Yuyang Cui, School of Computer and Communication, Lanzhou University of Technology, China*

ABSTRACT

In recent years, smart contracts have risen rapidly in the blockchain field, but security issues have also become increasingly prominent. Due to the lack of unified evaluation standards, the security analysis of smart contracts mainly relies on complex and not easily scalable expert rules. To address these issues, we employ slicing techniques to reduce the interference of extraneous code on the detection process, apply normalisation techniques to eliminate the differences between different compiler versions and use particle swarm optimisation algorithms to determine the similarity between contracts, thus improving the accuracy and efficiency of detection. In addition, we combine a variety of features such as static analysis, dynamic analysis and symbolic execution to gain a more comprehensive understanding of contract characteristics and behaviours for more accurate vulnerability identification. Experimental results show that the scheme significantly improves the detection capability and provides a new solution for the security detection of smart contracts.

KEYWORDS

Graph Embedding Algorithm, Multimodal Feature Fusion, Particle Swarm Optimisation Algorithm, Smart Contracts, Vulnerability Detection

INTRODUCTION

With the rapid development of blockchain technology, smart contracts, as its core component, have been widely used in the fields of finance, supply chain management, and digital asset trading (Metz, 2021). However, the security of smart contracts has been an issue of great concern because smart contracts cannot be changed once they are deployed on the blockchain, and they may involve large amounts of money and essential business logic. Therefore, smart contract vulnerability detection has become an important topic in current research.

Although there have been some studies on smart contract vulnerabilities, the existing vulnerability detection methods still have some limitations due to the complexity of smart contracts and the specificity of blockchain. Traditional software vulnerability detection techniques are usually not

DOI: 10.4018/IJSWIS.342850 *Corresponding Author

directly applicable to smart contracts because the execution environment of smart contracts is very different from ordinary software (Lu et al., 2021). Therefore, new vulnerability detection methods and tools need to be developed for the characteristics of smart contracts.

In recent years, researchers have proposed a number of smart contract vulnerability detection methods based on techniques such as static analysis, dynamic analysis, and symbolic execution (Nguyen et al., 2021). These methods can help developers discover potential vulnerabilities and provide remediation suggestions before deploying smart contracts. In addition, some research has been devoted to developing smart contract vulnerability detection tools to improve the efficiency and accuracy of vulnerability detection (Alweshah et al., 2020; Nedjah et al., 2023). However, smart contract vulnerability detection still faces challenges. The complexity of smart contracts and the decentralized nature of blockchain increase the difficulty of vulnerability detection and make it challenging to ensure the completeness and accuracy of detection. Therefore, improving the efficiency and reliability of smart contract vulnerability detection is still one of the pressing issues in current research. Although the match-based approach has been proven to be effective for vulnerability detection, applying the technique to smart contracts is a challenging task (Fatemidokht et al., 2021). Two significant issues need to be addressed: First, research has shown that the match-based detection technique should be applied more to bytecode since few smart contracts are open source. However, due to the rapid development of the Solidity compiler (Kumar & Sivakumar, 2022), the same bytecode fragment can produce different bytecodes depending on the compiler version, and this diversity interferes with bytecode matching. Another problem is that different versions of compilers can compile many different instructions, resulting in missing instructions. Even if the instructions have the same semantics, different compiler versions can cause significant differences.

To address these problems, this paper first sliced and normalized the program. Specifically, this paper employed slicing techniques to effectively reduce the interference of extraneous code on the analysis results. At the same time, they also adopted normalization techniques to eliminate the differences that different compiler versions may cause. On this basis, they designed unsupervised graph embedding algorithms to capture the structural information of functions. Through these methods, it is possible to significantly reduce the diversity of bytecode generation and the interference of noisy codes due to the rapid development of compilers, which in turn significantly improves the accuracy and effectiveness of detection. The normalized slices are mapped into vectors by an embedding graph network and then matched using a particle swarm optimization algorithm. The authors also introduced a multimodal feature fusion technique to improve detection accuracy. This enables the matching of contracts with different implementations and similar code logic (Raj & Pani, 2022). The authors' approach reduces the false alarm rate by more than 90% compared to direct matching via program slicing techniques. In addition, the number of detected vulnerabilities is significantly increased by applying standardization techniques. This approach effectively reduces the number of false positives and misses, while significantly improving the accuracy of code matching and successfully identifying undetected vulnerability contracts (Madan & Bhatia, 2021). By employing a multimodal feature fusion technique, the authors' approach significantly improves the accuracy of vulnerability detection.

Related Work

Because of the nature of blockchain, smart contracts are complex to fix once deployed. However, there has yet to be a complete library of innovative contract vulnerability features. Collecting all vulnerability logic and summarizing the corresponding features is difficult (Hu et al., 2022). In this regard, there is an intuitive and feasible solution: Treat known vulnerabilities as special features directly, turn the code to be detected into vectors and other easily comparable forms together, and match and eventually compare the similarity between the detection contract and the known vulnerability contract based on this and through particle swarm optimization (PSO) algorithm. If the similarities exceed the established threshold, the code that needs testing is suspect and should be reviewed further. It is worth mentioning that only a few smart contracts are currently open source. According to the

Figure 1. BeerCoin Contract

```
4   function creditEqually(address[] users, uint256 value) public
5       onlyOwner returns (bool){
6           uint256 balance = balances[msg.sender];
7           uint256 totalValue = users.length * value;
8           require(totalValue <= balance);
9           balances[msg.sender] = balance - totalValue;
10      }
```

authors' statistics, as many as 1.5 million smart contracts have been deployed on the blockchain; however, only about 2% (Nhi & Le, 2022) of the source code, that is, 32,499, is publicly viewable on the Etherscan browser. In addition, Etherscan has recently adapted its Web site functionality to no longer extend support direct retrieval of the source code of all published contracts, but only the 500 most recently published contracts, a change that further increases the difficulty of accessing the source code of contracts (Chawra & Gupta, 2022). Since most of the contracts are closed source code, it is possible to only analyze their bytecode. Therefore, the authors propose a smart contract vulnerability detection method based on byte matching to effectively detect potentially vulnerable contracts by calculating the similarity between the contracts to be detected and the known vulnerable contracts (Singh et al., 2022).

First, smart contracts will appear more homogeneous than traditional programs, which makes the matching effort more vulnerable to attacks by code segments unrelated to the vulnerability logic, the so-called "noise" (Madhumala & Tiwari, 2022). In the vulnerability contract shown in Figure 1, only lines 3 and 4 of the code are relevant to the vulnerability logic. At the same time, the rest of the code is noisy and irrelevant to the vulnerability logic.

When used as a "seed" for a match, it can mistakenly capture many contracts that bear only a high similarity to noisy code, leading to many false positive results.

Secondly, the source code for the smart contract, compiled into bytecode by the Solidity compiler, is now available in dozens of versions (Sissodia et al., 2022).

In different compiler versions, bytecode is generated in very different ways. Even if the source code is the same, the bytecode generated by different compiler versions varies widely. The diversity of compilers hinders matching efforts in many ways. Even if a potential vulnerability contract is almost identical to a known vulnerability contract, different compiler versions will likely result in differences in bytecode, reducing the similarity between them and ultimately leading to missed reports (He et al., 2020).

Third, in code matching, codes are converted into quantifiable, measurable, and comparable forms to facilitate similarity metrics. For example, Yamaguchi et al. (2011), utilizing a bag of features (Nguyen et al., 2020), encoded the feature information of each function as a vector. While this work has yielded satisfactory results, we need to consider functional structure information in greater depth to ensure that reporting in the area of smart contracts can be more complete. The current lack of attention to functional structure information may make the content of reports in this area appear to be lacking.

To solve the above problem, the authors first use a program slicing technique to reduce the effects of noise. Before matching, the contract's bytecode is sliced into thin slices, with each slice describing the instructions that the data went through. The slices most likely to reflect the core logic of the vulnerability are left for better results. Next, the authors normalize the values, adjust the order of the parameters, and remove unimportant instructions, so that contracts that are similar at the source level remain as similar as possible at the bytecode level, improving the accuracy of the matching. Then, they train an embedding graph network to obtain a vector representation of all normalized slices.

Finally, a PSO algorithm calculates the similarity between the slice to be tested and the vulnerability slice (Parizi et al., 2018).

Contributions

The main contributions of this paper are as follows:

1. The authors innovated the slicing and normalization process of smart contracts, by which the interference of noisy code is reduced and the accuracy of code matching is improved. In addition, the number of detected vulnerabilities increases substantially after applying the normalization technique.
2. The authors applied the PSO algorithm to smart contract vulnerability detection, which effectively reduces the number of false alarms and missed alarms by using known vulnerable contracts to find similar vulnerable contracts.
3. The authors introduced a multimodal feature fusion technique for smart contract vulnerability detection, which combines multiple features, such as static analysis, dynamic analysis and symbolic execution, to provide a more comprehensive understanding of the contract's characteristics and behaviors in order to improve the accuracy of vulnerability detection. According to the authors' observation, the detection effect on most smart contracts is very significant.

MATERIALS AND METHODS

Materials

In this section, the authors will give a brief overview of the background of their research, which includes several types of significant vulnerabilities in smart contracts, the data processing mechanism of the Ethernet virtual machine (EVM), and the PSO algorithm and multimodal feature fusion techniques we use.

In this paper, the authors mainly use these two techniques to detect vulnerabilities. They will focus on the following five categories of vulnerabilities: Reentrant vulnerabilities, integer overflow vulnerabilities, nasty random source vulnerabilities, access control vulnerabilities, and undetected return value vulnerabilities. All of these vulnerabilities are invariably related to improper manipulation of insecure data. A brief description of each vulnerability and detection methodology will follow.

Reentry Vulnerability

In Ethernet, smart contracts can call each other externally. Ethernet users can execute smart contracts and send Ethernet coins to the receiving account address, either an external account or a contract account, as both types of accounts can be transferred and otherwise manipulated. There is a special fallback() function in contract accounts, which has the main form of no function name and no parameters. When the user sends the etheric to the contract account in the etheric, the fallback() function in the contract account is automatically activated. Once transferred to the contract address set by the attacker, the fallback() function of the attack contract is forced to execute.

An attacker can add malicious code to the fallback() function and call the transport code of the attacker's contract again, causing damage to the user.

For example, the "TheDAOincident" (Wu et al., 2021) that previously shocked Ethernet is a typical example of hackers using a reentryable vulnerability to attack a contract.

Integer Overflow Vulnerability

An attacker can damage a user by adding malicious code to the flllback0 function and repeatedly calling the transport code of the attacker's contract.

Figure 2. Bad Random Source Vulnerability

```
5   function play() public payable{
6     requir(msg.value >= 1 ether)
7     if(keccake256(timestamp) % 2 == 0){
8       msg.sender.transfer(1.9 ether);
9     }
10  }
```

For example, unsigned integer uint8 is stored in the range of 0~255, (uint8)255+1 appears. The upper overflow is incorrectly stored as 0, and the lower overflow of (uint8)0-1 is incorrectly stored as 255. Also, signed integers can have overflow or underflow (Wang et al., 2019).

EVM specifies a fixed size for the data type. When the data exceed the specified range, an overflow will occur, leading to unpredictable losses if exploited by an attacker.

Bad Random Source Vulnerability

As Figure 2 shows, there is a nasty random source vulnerability in the betting contract.

The contract authors use timestamps as a random source in the third exercise to determine the outcome of the betting game by generating random numbers. However, finding a suitable source of randomness in Ether takes work (Zhou et al., 2014). Although it is impossible to predict the timestamp at the time of execution accurately, an attacker could test whether the condition in line 3 would hold by deploying another contract that performs the same calculation and immediately engaging in a bet when the test is actual, thus becoming invincible. Thus, data on blocks (including timestamps) are undesirable sources of randomness and using them to generate random numbers poses a security risk.

Access Control Vulnerabilities

Typically, the functions used to set the owner of a contract should be privileged functions or constructors that ordinary users cannot call. However, in some contracts, everyone has access to these functions, resulting in everyone being able to be the owner of the contract and thus able to perform certain privileged operations (Zhou et al., 2020).

Undetected Return Value Vulnerability

Ethernet smart contracts provide interfaces such as delegate call, call, send, all code, and other underlying functions that can be checked for successful execution by the function's return value.

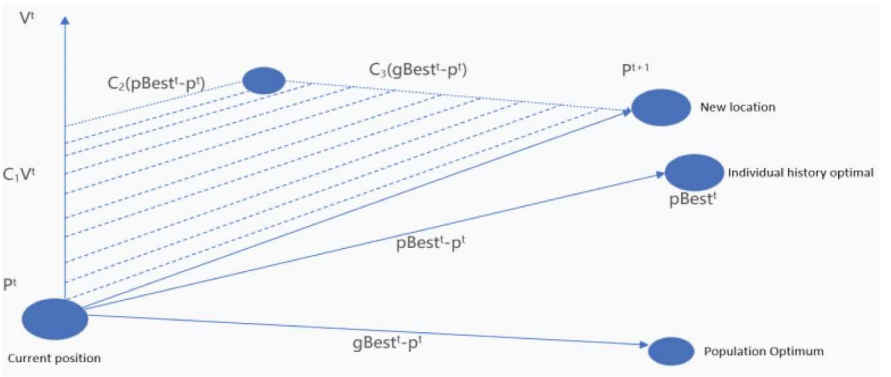
If the return value is not determined, the program will continue to execute, which may lead to serious consequences. The features are summarized as follows:

1. Smart contracts use delegtecall(), call(), send(), callcode(), and other underlying functions.
2. The return value is not checked and processed, and the subsequent operation is continued.

Adaptation Value Function

The fitness function evaluates the performance of each particle to ensure the timely elimination of particles with poor performance. Previous studies have determined whether a suspicious smart contract has contractual defects by studying the node information in the embedding vectors and transforming the contract slices into embedding vectors to determine whether the suspicious smart contract is similar to a known vulnerable contract.

Figure 3. Conceptual Diagram of the Particle Swarm



TLSaaty (Hamp-Lyons, 1990) of the University of Pittsburgh proposed the analytical hierarchy process (AHP) in the 1970s, which requires decision-makers to determine the priority or weight of each attribute by comparing the relative importance of each attribute.

Particle Swarm Optimization Algorithm

The PSO algorithm is a stochastic search algorithm proposed by James Kennedy and Russell Eberhart (Yang et al., 1999) in 1995, inspired by the behavior of flocks of birds in nature. In the search space, each particle independently searches for the optimal solution and records it as an individual extreme value. Then, all particles share the individual extremes and use the found individual extremes as the global optimal solution. According to the global optimal solution, all particles will adjust their speed and position. The algorithm mainly includes the steps of initializing the particle swarm, evaluating the particles, finding the individual extreme values, finding the global optimal solution and adjusting the speed and position of the particles. Figure 3 shows the conceptual diagram of the model.

Multimodal Feature Fusion

Multimodal feature fusion technology is a method of fusing data from different modalities to obtain more comprehensive, accurate, and reliable information. In the multimodal feature fusion process, data from different modalities are extracted with their respective features, and then these features are combined to achieve a comprehensive integration of information.

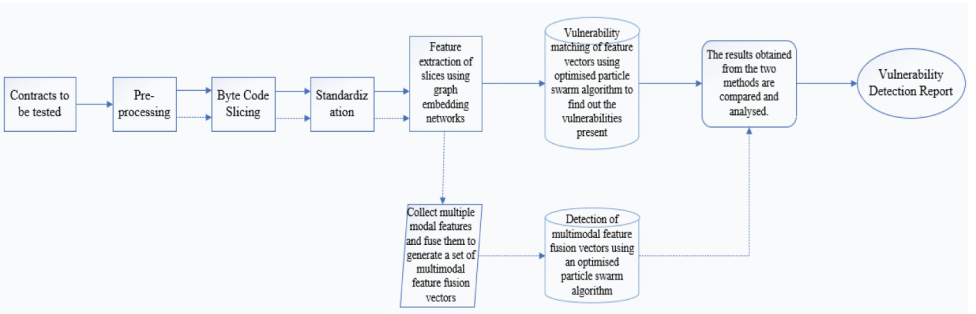
The multimodal feature fusion technique can be applied in several domains, for example, in smart contract vulnerability detection, by integrating multiple features such as static analysis, dynamic analysis, and symbolic execution, which enhance the accuracy of vulnerability detection. Static analysis can precisely extract syntactic and structural information from the code, dynamic analysis can capture the running behavior and interactions of the program, and symbolic execution can simulate the execution process of the program and capture possible errors.

In addition to smart contract vulnerability detection, multimodal feature fusion techniques can be applied to other areas, such as natural language processing, computer vision, and multimedia processing.

METHODS AND SOLUTION MODEL

The general idea of the method is to extract patterns from the contracts to be tested and match them with those in the contracts with known vulnerabilities using a PSO algorithm. However, due to the

Figure 4. Flowchart of Contract Detection



small size of the smart contract, any modification can hinder the matching process. Figure 4 shows the flowchart of this approach.

1. **Preprocessing:** The processing contract and the known vulnerability contract are preprocessed to convert the contract bytecode into instruction set form and construct the control graph.
2. **Control Flow Graph (CFG) Traversal:** Traverse the CFG of the smart contract, find and mark all the specific instructions as slicing conditions, and complete the slicing operation by simulating the execution of the contract to confirm the relationship between data and instructions.
3. **Normalisation:** Normalise the obtained slices to reduce bytecode differences caused by compiling with different versions of compilers.
4. **PSO Algorithm Matching:** A vector representation of all normalized slices is obtained by training the graph embedding network. Finally, the similarity between the slices to be tested and the vulnerability slices is calculated using a PSO algorithm.
5. **Multimodal Feature Fusion:** Based on the above work, the authors further introduce a multimodal feature fusion technique to improve the detection accuracy of vulnerability contracts. Specifically, the authors fuse three features, namely, static analysis, dynamic analysis, and symbolic execution, to form multimodal feature vectors. These multimodal feature vectors describe the characteristics and behaviors of the contract more comprehensively, thus improving the accuracy of vulnerability detection. These feature vectors are selected based on their ability to effectively describe the characteristics and behaviors of the contract and their high sensitivity and specificity for discovering potential vulnerabilities.
6. **Comparison Results:** Using the fused modal eigenvectors, the PSO algorithm or other vulnerability detection techniques are again applied for vulnerability detection.

   This can reveal more potential vulnerabilities or further confirm previously detected vulnerabilities.

Algorithm Configuration

Particle Swarm Optimization Algorithm Configuration. The PSO algorithm is a group intelligence-based optimization tool that finds the optimal solution to a problem by simulating the behavior of a flock of birds foraging for food. In smart contract vulnerability detection, the PSO algorithm is used to search for the most similar contract to a known vulnerable contract in the feature space.

- **Particle Encoding:** Each particle represents a potential solution, that is, a feature vector of a smart contract. Graph embedding algorithms and multimodal feature fusion techniques generate these feature vectors.

- **Fitness Function:** The fitness function is used to evaluate the strengths and weaknesses of each particle, that is, its similarity to a contract with known vulnerabilities. Metrics such as cosine similarity and Euclidean distance can be used here.
- **Velocity and Position Update:** The particle updates its velocity and position based on its historical optimal position and global optimal position to search for better solutions in the feature space.

Multimodal Feature Fusion Configuration. The multimodal feature fusion technique is used to combine features from different sources and types to provide a more comprehensive and accurate description of the contract.

- **Feature Extraction:** Extract multiple types of features from smart contracts, including static code features (e.g., function call graphs, and control flow graphs), dynamic execution features (e.g., transaction behaviors and resource consumption), and symbolic execution features (e.g., path conditions and reachability states).
- **Feature Fusion:** Use appropriate fusion strategies (e.g., weighted sum, concatenation, and deep learning models) to fuse these features into a unified feature vector for subsequent analysis and comparison.

Parameter Selection

When configuring the algorithm, the selection of parameters has a significant impact on its performance and results. The following are some key parameters and their selection considerations:

- **Particle Swarm Size:** The size of the particle swarm determines the degree of coverage of the search space. A too small particle swarm may lead to inadequate search, while a too large particle swarm may increase the computational overhead. It needs to be chosen reasonably according to the actual problem and computational resources.
- **Learning Factor:** The learning factor is used to control the speed of the particles to learn from their own historical and global optimization. A more prominent learning factor may accelerate the convergence but may also lead to a premature fall into the local optimum; a smaller learning factor may make the search process more robust, but the convergence speed is slower.
- **Inertia Weights:** Inertia weights determine how well the particle velocity is maintained. Larger inertia weights favor global search, while smaller inertia weights favor local search. Inertia weights can be adjusted to balance the ability of global and local search.
- **Feature Weights:** In multimodal feature fusion, different features may have different degrees of influence on the results. Appropriate weights need to be assigned to each feature according to the actual situation to ensure that the fused feature vectors can accurately reflect the characteristics of the contract.

Contract Slicing

Slicing Conditions. Any data from outside the contract may be directly or indirectly controlled by an attacker or easily known by an attacker, who can use it to attack the smart contract. With this in mind, we use all instructions to import data from outside as slicing conditions. Specifically, the slicing condition may introduce the following four types of data:

1. **Transaction Data:** The transaction data are provided directly by the caller and are introduced for execution by the CALLDATALOAD and CALLVALUE instructions.
2. **Data on the Block:** In EVM, everyone can easily access the data on the blocks. An attacker can break the randomness by simulating the same random number calculation process if they are used as a random source. Instructions to introduce data on the block include BLOCKHASH and TIMESTAMP.

3. **External Data:** The data stored in external memory usually play a vital role in the contract and can be accessed by anyone (e.g., the user's balance data). The command SLOAD (SafeMath, 2022) retrieves data from external storage.
4. **The Return Value of an External Call:** The caller contract should explicitly verify the return value of the external call. Instructions to execute external calls include CALL and SEND. The authors will first traverse the smart contract once, marking all the instructions in the above range as a basis for subsequent slicing.

Model Execution and Slicing. In the bytecode of a smart contract, the vast majority of instructions interact with data through the stack, memory, and external memory, and the association between instructions and data is not directly reflected in the bytecode except for push instructions. However, slicing requires analysis of the dependencies between instructions and data, and to confirm such dependencies, the authors choose to slice the smart contract based on simulated execution.

Normalization

In this section, the authors further normalize the generated slices to reduce bytecode differences caused by compiling different compiler versions. Figure 5 shows the general flowchart of slicing and normalization.

Numerical Normalization. First, the specific values of the data are replaced with labels representing data attributes. The purpose of this step is to characterize the data better. Constants are widely used in intelligent contract bytecode, such as the target address of jump instructions and offsets to memory addresses. Using different compiler versions may change these data, introducing unwanted differences in our slices. Furthermore, it is challenging to characterize data by specific values alone. By using information such as where the data came from and which instructions it went through, it is possible to make the data more distinctive and, therefore, the instructions more easily distinguishable. The authors characterize values by labelling the data and allowing the labels to propagate through the execution. The label of the data depends on the instruction it obey, and Table 1 shows the labels that some instructions will assign to the resulting data.

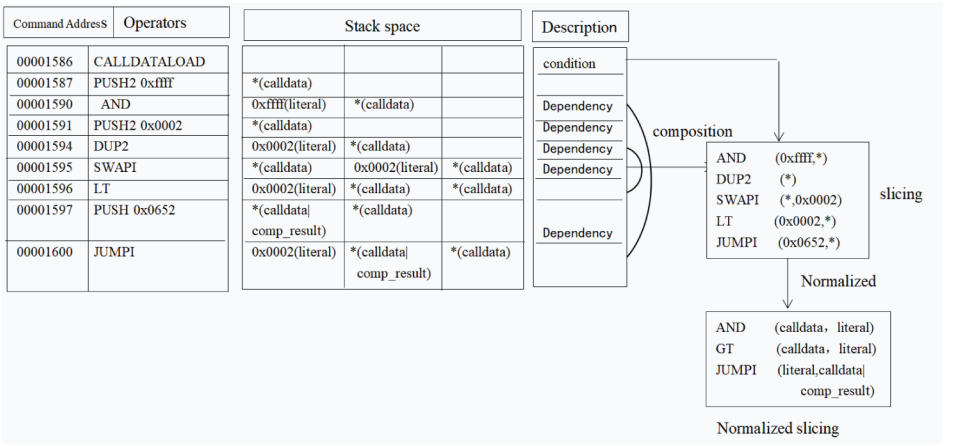
Parameter Data Adjustment. When versions of the Solidity compiler are upgraded, there may be differences between the old and new versions, which may cause them to generate different bytecodes for the same source code. For example, in Solidity v0.4.22 (Xu et al., 2019), if a SWAP1 directive precedes a comparison operator (e.g., LT and GT), the compiler replaces the comparison operator with its opposite operator. It removes the SWAP1 directive (Ni et al., 2020).

To reduce this unnecessary discrepancy, the authors first sort all labels of data in dictionary order; then, the arguments of some multiparameter instructions are also sorted in the dictionary order of their attributes. In this process, if the order of the arguments of the comparison operators changes, they are replaced by the opposite operators accordingly. Table 1 shows the descriptions involved.

Table 1. De Novo Ordering of Parameter Order

Instruction	Adjustment method
lt,gt,slt,sgt	Instruction (attrB,attrA) → opposite instruction (attrA,attrB)
ADD,MUL,EQ,	Instruction (attrB,attrA) → Instruction (attrA,attrB)
OR, AND, XOR	

Figure 5. Example of Slicing and Normalization



Removing Irrelevant Commands. In EVM, there are two classes of instructions that operate directly on data in the stack: The DOP class and the SWAP class. Due to the different versions, these instructions can be converted to each other. This variation is not conducive to matching. Moreover, although DOP, SWAP, and POP instructions operate on data in the stack, they may be more relevant to source code behavior. The authors consider their role in code matching to be negligible, so instructions related to DOP, SWAP, and POP will be removed from the slice.

The authors briefly demonstrate the effect of slicing and normalization with the example in Figure 5. In this example, they choose the first instruction, CALLD ATALOAD, as the slicing condition and slice it by analyzing which instructions have data dependencies on them.

Figure Embedding. After slicing and normalization, the authors get relatively pure and homogeneous slices. At this point, they need to embed the slices into the vector space to measure their similarity. Yamaguchi et al. (2012) used a bag-of-features encoding method to map functions into vectors based on the extracted features. This encoding method assumes that all features are of an independent dimension, which leads to the relationship between features, that is, the structural information of the function, being ignored. In smart contracts, many instructions are strongly correlated, and these correlations can provide richer semantic information (e.g., instructions that execute external calls are correlated with instructions that read or write external data). Conditional judgment instructions usually control instructions that terminate execution.

Suppose two functions use different methods to compute the total amount of transfers, resulting in the instructions in their slices being very different. In that case, the direct use of the bag-of-features approach cannot accurately reflect their similarity. In order to better capture the structural information in the slices, the authors convert the slices into the form of program dependency graphs. By using a graph embedding network, the researchers can compute the embedding vectors of the slices, which better prepares for subsequent tests of the PSO algorithm and multimodal feature fusion techniques. In this way, it is possible to compare the similarity of the two functions more accurately and improve the accuracy of smart contract vulnerability detection.

Algorithm Overview. IGraph2Vec is an algorithm for unsupervised graph filtering based on root subgraph extraction, which can effectively preserve the structural information of the graph.

Algorithm 1. Graph embedding network

Input: smart contract C={G_1, G_2, ... }

Embed Big Little S

Output: the G-embedding vector for each slice$\boldsymbol{v}_G \in R^S$

2. *Randomly initialize* $\boldsymbol{v}_G$ *FOREACHG* ∈ C

3. ROR $G_i \in$ C:

4. SG_n = *SubGraph*(n, G_i) *FOREACH* $n \in N_i$

4. FOR$G_i \in C$

11. $J(\lambda, P) = -\log \sum_{n \in N_i} P_r(SG_n | \boldsymbol{v}_{G_i})$

12. $\lambda = \lambda - \alpha \cdot \partial J / \partial \lambda$

13. $P = P - \alpha \cdot \partial J / \partial P$

14. FOR $G_i \in C$

15. $J(\boldsymbol{v}_{G_i}) = -\log \sum_{n \in N_i} P_r(SG_n | \boldsymbol{v}_{G_i})$

16. $\boldsymbol{v}_{G_i}$ == *sync, corrected by elderman* == $\boldsymbol{v}_{G_i} - \alpha - \partial J / \partial \boldsymbol{v}_{G_i}$

In this paper, the authors refer to the algorithm of Graph2Vec (Sayeed et al., 2020) to embed vectors. They map the operators in the instructions directly to vectors with data labels and use these vectors as attributes of the nodes that participate in generating the root subgraph vectors. The graph embedding network aims to map all slices in a smart contract into vectors. Thus, the input to the graph embedding network is the smart contract C containing the slices to be embedded and the dimension S (Xing et al., 2020) of the embedded vectors. At the same time, the output is the S-dimensional vector obtained after embedding each slice. The process of the graph embedding network is as follows: First, the embedding vectors of all slices appearing in the contract C are randomly initialized, and the embedding vectors of all root subgraphs in these slices are computed; then, two stages of training are performed, that is, in the first stage, the parameters affecting the embedding of the root subgraphs are trained, and in the second stage, these parameters are fixed, and the embedding vectors of the slices are trained and output.

For a smart contract C = { G_1, G_2, ..., G_n }, G_i is the program dependency graph of the slices extracted from that contract, defined as G_i = { N_i, E_i, λ}, where N_i is the set of instructions contained in the slice, E_i is the dependency between instructions, and λ represents a mapping that maps the instructions n to an S-dimensional vector based on the properties of the operators and data of the instructions in the slice v_n. Algorithm 1 shows the flow of the graph embedding network.

First, the embedding vectors of all slices are initialized randomly. Lines 2 to 3 of Algorithm 1 represent the computation of the embedding vector of the root subgraph of each instruction node in

all slices of the contract C. The root subgraph embedding vector in line 3 of Algorithm 1 is computed as shown in Equation 1 (Momeni et al., 2019):

$$SubGraph(n,G_i) = \tanh\left(v_n + P\sum_{m\in Neib(n,G_i)} v_m\right) \quad (1)$$

where P is the parameter matrix, Neib(n,) is the neighbor node of node n in graph Gi. Next, the authors will obtain the embedding vectors of the slices in two stages, which are reflected in lines 5 to 10 of Algorithm 1. Equation 2 shows the probability function used for this process.

$$\Pr(SG_n \mid v_{G_i}) = \frac{\exp(v_{G_i} \cdot SG_n)}{\sum_{w\in N_c} \exp(v_{G_i} \cdot SG_w)} \quad (2)$$

where N_C denotes the concatenated set of nodes of all slices in contract C. Equation 2 calculates the root subgraph Sappears in G_i in a slice, and the goal of training is to maximize it. As mentioned before, the authors will perform two stages of training: In the first stage, they focus on λ and P and train only on these two parameters; in the second stage, they keep the parameters λ and P unchanged and train on the embedding vector of slices. In this way, they provide a data source for comparison detection using the PSO algorithm in the following.

Particle Swarm Optimization Algorithm to Detect Vulnerable Contracts

Optimizing Particle Algorithm Configuration and Parameter Selection.

a. Algorithm Configuration.
b. Particle Swarm Initialization:
c. Number of Particles: Select the number of particles as 50; this number can provide enough search space while maintaining computational efficiency.
d. Particle Dimension: Set the particle dimension according to the number of parameters to be optimized. For example, to optimize three parameters, namely, Learning rate, regularization factor, and iteration number, the particle dimension is 3.
e. Initial Position and Velocity: Randomly initialize the position and velocity of the particles within the allowed range of the parameters.
f. Adaptation Function: As to evaluation metrics, the combination of accuracy rate (ACC) and false positive rate (FPR) is used as the fitness function.
g. Particle Update Strategy: Use the standard PSO velocity and position update formula to update the velocity and position of particles. Each particle records its historical optimal position (individual optimal) and the optimal position in the whole population (global optimal).
h. Parameter Selection.

1. **Inertia Weights:** It includes the following features:
   a. **Initial Inertia Weights:** A sizeable initial value (e.g., 0.9) is chosen to maintain an extensive exploration capability at the beginning of the search.
   b. **Decay of Inertia Weights:** Gradually reduce the inertia weights as the number of iterations increases to facilitate the convergence of the algorithm. Linear decay or nonlinear decay can be used.

2. **Acceleration Coefficients:** Both the individual acceleration coefficients and the overall acceleration coefficients were set to common values. This helps to balance the effects of individual experience and group experience on particle motion.
3. **Maximum Velocity and Position Limits:** Appropriate maximum velocity limits are set according to the definition domain of the parameter to prevent the excessive spreading of particles in the search space. The position limit can be set according to the value range of the parameter to ensure that the particles will not go beyond the legal range of the parameter.
4. **Termination Conditions:** It includes the following features:
    a. **Maximum Number of Iterations:** Set an appropriate maximum number of iterations (e.g., 100 or adjust according to the complexity of the problem).
    b. **Adaptation Threshold:** Set a higher adaptation threshold; when the adaptation of the particle reaches or exceeds this threshold, the algorithm can be terminated early.

Adaptation Value Function

The fitness function is a function used in evolutionary algorithms to measure the degree of merit of an individual in the solution space. It is usually designed as a mathematical function that accepts a candidate solution (also known as an individual) as input and returns a numerical value indicating the degree of fitness of that solution. In the optimization process of an evolutionary algorithm, the goal of the fitness function is to maximize or minimize this value in order to find the optimal or suboptimal solution.

The design of the fitness function usually depends on the nature and objective of the particular problem. In the scenario of smart contract security vulnerability detection, the fitness function may be defined based on the characteristics of the contract and the security criteria.

Building the AHP Model. the first layer of decision-making should be determined; the first layer is the general objective of the decision, the second layer is the layer of different evaluation indicators, that is, attributes, and the third layer is the layer of alternatives of the decision. This paragraph studies the similarity problem of the embedded vectors of slices, so the general objective of the first layer of decision-making should be the similarity of the nodes of the embedded vectors (Liu et al., 2018). There are many evaluation indexes in the second layer to judge whether the nodes are similar or not, and after comparing and synthesizing, the authors divide the evaluation indexes of similarity into the following three categories:

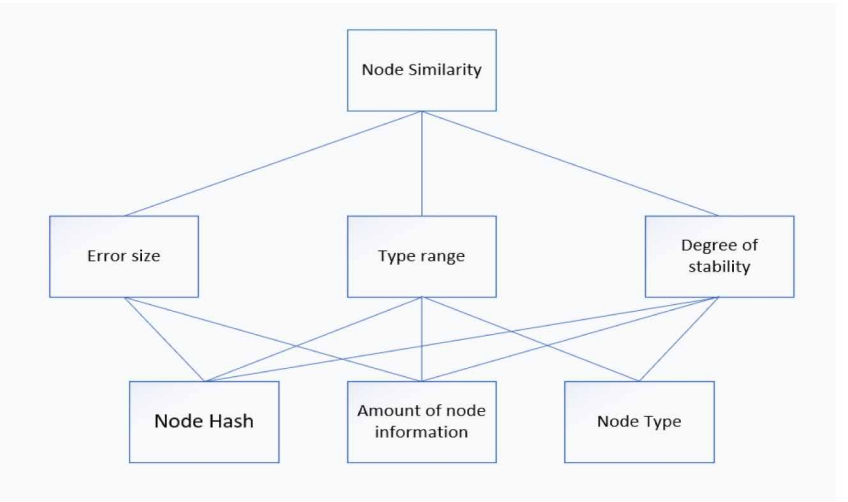
1. **Error Size:** There may be errors in the embedding vector analysis of the root subgraph; set this error value.
2. **Type Range:** The degree of support for different node types.
3. **Stability:** Whether the similarity can genuinely reflect the similarity of the nodes in case of significant differences in similarity.

The third layer is the alternative; the authors chose three: Node hash value, node information amount, and node type. Figure 6 shows the AHP three-layer model in summary.

Setting the Judgment Matrix. It includes the following steps:

1. **Determine the Objective Function:** In smart contract vulnerability detection, the objective function can be detection accuracy, recall rate, and F1 score. The establishment of the objective function will directly affect the direction and focus of the PSO algorithm.
2. **Determine the Influencing Factors:** The results of smart contract vulnerability detection are affected by several factors, including the static features, dynamic features, and control flow features of the contract. The static features include the bytecode of the contract, as well as the control flow features. The static features include the byte code and function

Figure 6. AHP Three-Tier Model



call relationship of the contract; the dynamic features include the data flow and control flow information during the execution of the contract; the control flow features include the program flow and branch structure of the contract.

3. **Construct the Feature Library:** According to the determined influencing factors, extract the corresponding static features, dynamic features and control flow features from the smart contract to be tested. For example, specific opcodes can be extracted from bytecode, the reading and writing of variables can be analyzed from data flow, and critical jump points can be identified from control flow. These features will form a feature library, which will provide the basis for subsequent vulnerability detection.
4. **Constructing a Judgment Matrix:** For each feature, it is necessary to compare it with other features to get the judgment matrix. The judgment matrix is a square matrix in which each element represents the relative importance of one feature compared to another. When constructing the judgment matrix, the interrelationship and importance of different features need to be considered to ensure the consistency and accuracy of the judgment matrix. The relative importance of information between features can be obtained through expert assessment and historical data analysis.
5. **Determine Feature Weights:** By calculating the feature vector of the judgment matrix, the weight vector of each feature can be obtained. Each element of the feature weight vector indicates the degree of importance of the corresponding feature in the objective function.
6. **PSO Algorithm:** This method determines the optimal solution by using a PSO algorithm to optimize the weight vector corresponding to each feature as an objective function. In the specific implementation, it is necessary to first randomly initialize each particle in the particle swarm, including their positions and velocities. Then, the adaptation value of each particle is computed according to the objective function. During each iteration, each particle compares its current adaptation value with the adaptation value of the best location in history. If the current adaptation value is better, it is taken as the current optimal solution. At the same time, the entire swarm updates its optimal solution to the best of the historical optimal solutions of all the particles based on the optimal solution of the current particle and the optimal solution of the entire swarm, as well as the velocity and position information of the particles to renew the speed and position of the particles. This process repeats until the

end conditions are met, such as reaching the predefined maximum number of iterations or finding a sufficiently good solution.

7. **Detecting Vulnerabilities:** Using the obtained optimal solution (i.e., feature weight vector), the smart contract to be tested is detected for vulnerabilities. Precisely, the smart contract can be executed to collect its feature data during execution and weight the feature data according to the optimal solution to obtain a comprehensive score. Based on this score, whether the smart contract to be tested has vulnerabilities or potential risks is judged.

Consistency Test. Since the judgment matrix obtained by manual two-by-two comparison is highly subjective, we need to conduct consistency tests on it to ensure the accuracy of the results. The consistency test can increase the subjectivity of the judgment matrix and the credibility of the results.

First, find the maximum characteristic root of each judgment matrix. $\lambda_{\max}$ is the largest characteristic root:

$$\lambda_{\max} = \sum_{i=1}^{n} \frac{\left(Aw\right)_i}{nw_i} \tag{3}$$

Second, calculate the consistency index CI by maximum characteristic root:

$$CI = \frac{\lambda_{\max}}{n-1} \tag{4}$$

where W is the matrix normalized, the matrix after linear transformation, n denotes the order of the comparison judgment matrix and is the maximum characteristic root of the judgment matrix.

Third, determine the corresponding evaluation random consistency index RI.

According to the order of the judgment matrix, the corresponding average random consistency index value can be obtained by looking up the table. The judgment matrix of this paper is of the third order, so the corresponding RI=0.52

Finally, calculate the consistency ratio CR:

$$CR = \frac{CI}{RI} \tag{5}$$

In the ideal case, CR is 0. However, in the actual evaluation, subjective factors lead to errors purely in, so, when CR < 0.1, the difference is small enough to pass the test although the ideal case is not satisfied.

The final consistency test was satisfied by the above matrices: 0.0922, 0.0176, 0.0089, and 0.0176, respectively.

Determination of Attribute Weights. After getting the judgment matrix, it is possible to determine the weights of the attributes based on the judgment matrix. The steps are as follows.

First, the vector is normalized. Assume that the judgment matrix $A = aij_{nn}$. After normalization, the matrix is $R = \left(r_{ij}\right)_{nn}$:

Table 2. Judgment Matrix of G1, G2, and G3 on the Acquaintance A

G1	Single sort weights	G2	Single sort weights	G3	Single sort weights	A	Single sort weights
P1	0.04	P1	0.14	P1	0.34	G1	0.62
P2	0.21	P2	0.62	P2	0.16	G2	0.24
P3	0.75	P3	0.24	P3	0.34	G3	0.14

$$r_{ij} = \frac{a_{ij}}{\sqrt{\sum_{i=1}^{n} aij^2}} \tag{6}$$

After the vectors are normalized to one, the $r_{ij} \in [0,1]$ and for each attribute index, the column vector is modulo 1.

Secondly, the normalized R matrix is summed by rows:

$$\overline{w_i} = \frac{\overline{w_i}}{\sqrt{\sum_{i=1}^{n} \overline{w_i}^2}} \tag{7}$$

Third, the vector $\overline{w_i} = \left[\overline{w_1}, \overline{w_2}, \overline{w_3} ... \overline{w_n}\right]^T$ is normalized:

$$w_i = \frac{\overline{w_i}}{\sqrt{\sum_{i=1}^{n} \overline{w_i}^2}} \tag{8}$$

After calculating, the hierarchical single ranking weights are obtained (Table 2).

The single ranking weight of the second layer elements with respect to the total objective of the first layer is obtained in the table $\left[0.62, 0.24, 0.14\right]^T$ denoted as $w_1^{k-1}(k=3, i=1,2,3)$. The single ranking weight of the third layer relative to the second layer is a 3*3 matrix, denoted as P_{ij}^k (k=3,i=1,2,3,j=1,2,3) .Then, the total ranking of the elements of the third layer with respect to the total objective is:

$$w_i^k = \sum_{j=1}^{3} P_{ij}^k w_j^{k-1} (i = 1,2,3) \tag{9}$$

Table 3 shows the calculated total ranking weights.

Table 3. Total Judgment Matrix of G1, G2, and G3 on the Acquaintance A

A	P1	P2	P3
Total ranking weights	0.106	0.301	0.570

Algorithm 2. Process of PSO

Input: The embedding vector of the two slices
Step 1: Initializing a swarm of particles initializes a set of particles (i.e., solutions) and their velocities based on the number and nature of feature vectors. Each particle represents a possible vulnerability detection result.
Step 2: A fitness value is computed for each particle, which indicates the quality of its corresponding solution. The fitness value may be defined according to practical needs; for example, it may be a vulnerability detection accuracy, a recall rate, or other evaluation metrics.
Step 3: Iterative evolution.
During each iteration, the particle updates itself according to two "poles:" The best solution currently found by the particle, called the particle pole best value, and the best solution currently found by the whole population, called the global pole best value. During the update process, the fitness function of the particle is first compared with its best value. If fitness is less than pbest, then the current position of the particle is pbest, and then the pbest_ value and gbest_ value of each particle are compared. If the pbest_ value of a particle is less than the gbest_ value, then that particle is the particle corresponding to the gbest.
Step4: Updating the particle swarm.
Update the position and velocity of the particles based on the fitness value. Updating the position of a particle by comparing its fitness value gives a higher probability that a good solution will be retained.
Step 5: Iteration termination condition.
Output: Similarity of the embedding vectors of the two slices.

Thus, the expressions of node hash-based, node-information-based, node-type evaluation-based, and node-similarity-based adaptation value functions are:

$$F\left(X\right)=\sum_{i=0}^{n}\left(0.106sim_{1}\left(x_{1}\right)+0.301sim_{2}\left(x_{2}\right)+0.570sim_{3}\left(x_{3}\right)\right) \tag{10}$$

Among them sim_1, sim_2 , sim_3 denotes the similarity value obtained from three evaluation methods based on hash value based on node information amount based on node type, and denotes the node serial number.

Algorithm Steps

Once the adaptive value function is derived from the AHP analysis, the algorithmic process can begin. The algorithmic process of PSO is shown in Algorithm 2.

Multimodal Feature Fusion Technology Continues to Detect Vulnerable Contracts

Based on the previous stage, the authors further use multimodal feature fusion to continue detecting vulnerabilities in smart contracts. Specifically, they first perform unimodal feature extraction to extract features from multiple aspects such as static analysis, dynamic analysis, and symbolic execution; then, they perform cross-modal alignment, and finally use multimodal feature fusion to generate a set of comprehensive feature vectors. The researchers then feed this set of feature vectors into the PSO algorithm to discover more potential vulnerabilities or further confirm previously detected vulnerabilities.

Unimodal Feature Extraction

Static Analysis Features. In static analysis feature extraction, for each vulnerability I, the authors obtain features such as vulnerability-related metrics and parameters that describe the intrinsic attributes and behaviors of the I. To perform feature extraction for each object, they use an average pooling operation to convert each object into a 4096-dimensional feature vector Oi, where i denotes a different object with a value ranging from 1 to m.

To further adjust the spatial weights in the feature map, the authors introduce an attention mechanism. By means of a linear projection layer, they transform each feature map fi into a 3D region feature Oi. The generation of these region features is based on the analysis of the smart contract source code or bytecode. Among the features that may be included are variable types, function calls, and control flow analysis:

$$O_i = W_o f_i + b_o, i \in [1, m] \tag{11}$$

where Wo and b_o are learnable parameters and C_i is the ith vulnerability-related features.

Dynamic Analysis of Features. In dynamic analysis feature extraction, the authors are given a dataset S and extract features such as frequency, duration, byte length, and data type from the input data. The extraction of these features is dynamically generated by the smart contract at runtime and thus has high temporal and spatial complexity.

To better capture these features, the authors use an average pooling operation to extract a 4096-dimensional feature vector, denoted as Si, for each input data. This feature vector is derived by averaging the various features of the input data, which effectively describes the global characteristics of the data.

However, these feature vectors do not take into account the behavior of the smart contract at runtime. Therefore, the authors further transform each feature map gi through a linear projection layer into 3D region features Si. These region features are generated at runtime of the smart contract and are able to capture the behavioral characteristics of the contract.

$$S_i = W_s g_i + b_s, i \in [1, p] \tag{12}$$

where W_s and b_s are learnable parameters and S_i is the ith data features.

Symbolic Execution Characteristics. In symbolic execution feature extraction, the authors first give a model M, which can be any model that describes the behavior of a smart contract. Then, they extract a variety of features from the model, including model structure, parameters, and training dataset. The extraction of these features can provide insight into the behavior of smart contracts and help us perform better vulnerability detection.

To better capture these features, the authors extract a 4096-dimensional feature vector, denoted Mi, for each model using a mean pooling operation. This feature vector is derived by averaging the various features of the model, which can effectively describe specific global or essential characteristics of the model.

However, these feature vectors do not take into account the execution behavior of the smart contract under various input conditions. Therefore, the authors further transform each feature map li into a 3D region feature Mi by means of a linear projection layer. These region features are generated by symbolic execution techniques, including reachability analysis and data flow analysis. This captures the behavioral characteristics of smart contracts under different input conditions:

$$M_i = W_m l_i + b_m, i \in [1, n] \tag{13}$$

where W_m and b_m are learnable parameters and M_i is the ith model features.

Cross-Modal Alignment

After the above unimodal feature extraction, a feature dictionary is created to map the features of each modality into a shared feature dictionary. This feature dictionary can be based on bag-of-words models or embedding techniques. Cross-modal alignment is achieved by mapping features of different modalities into the same feature space. The authors perform an alignment operation on the features of different modalities. This operation can be either timeline-based alignment or semantic-based alignment. For example, aligning the features of dynamic analysis and symbolic execution on the timeline can make them consistent on the timeline. In the course of their research, the authors found a complementary relationship between the features of dynamic analysis and symbolic execution, and, in order to strengthen the correlation relationship between the data and the model, they propose a cross-modal attention mechanism to align the modules, with the aim of aligning the vulnerabilities, the data, and the model in the smart contract in the embedding space. Given a set of vulnerability features R = {r1, ..., rm}, a set of data-level features T = {t1, ..., tp}, and model-level feature set W = {w1, ..., wn}. Inspired by Wang et al. (Datar et al., 2020), the region-word affinity matrix can be applied to the input image of a smart contract to assist vulnerability detection and analysis by matching regions in the image with variables or operators in the code to improve the accuracy and effectiveness of vulnerability detection:

$$A_1 = \left(\widehat{W}_o R\right)\left(\widehat{W}_t W\right) \tag{14}$$

$$A_2 = \left(\widehat{W}_s T\right)\left(\widehat{W}_t W\right) \tag{15}$$

where $\widehat{W}_o, \widehat{W}_s, and\ \widehat{W}_t$ and denote the projection matrices to obtain k-dimensional loopholes, data, and model features. For vulnerability-coding, data-coding affinity matrices $A_1 \in R^{m\times n}$, $A_2 \in R^{p\times n}$, Aij denote the affinity between the ith vulnerability and the jth coding and the affinity between the ith data and the jth coding, respectively.

In order to infer potential alignment between local segments from different schemas, this paper focuses on each segment of code in each region by further normalizing the affinity matrix A:

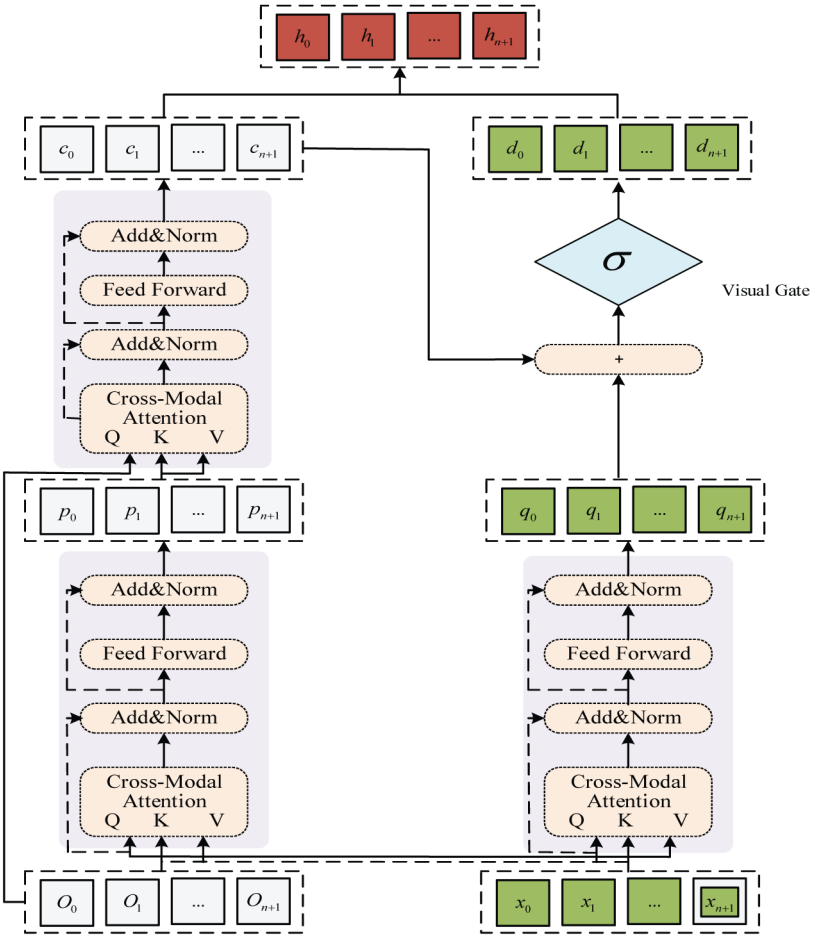
$$\overline{A_1} = soft\max\left(\frac{A_1}{\sqrt{k}}\right) \tag{16}$$

$$\overline{A_2} = soft\max\left(\frac{A_2}{\sqrt{k}}\right) \tag{17}$$

Then, all code features for each region are aggregated according to the normalization matrix $\overline{A_1}$, $\overline{A_2}$:

$$U_1 = \overline{A_1} \cdot W \tag{18}$$

Figure 7. Cross-Modal Alignment Module



$$U_2 = \overline{A_2} \cdot W \tag{19}$$

Row i of U_1 and U_2 characterizes the interactive model features corresponding to the ith region.

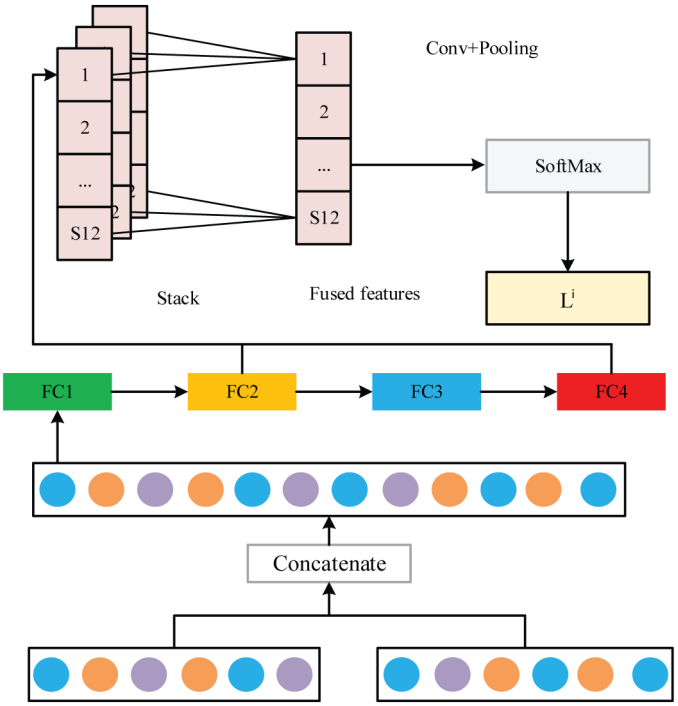
Multimodal Feature Fusion

Inspired by Tikhomirov et al (Tikhomirov et al., 2018), in this study the authors used a mid-term integration approach to fuse multimodal features by stacking pool modules. Figure 8 shows the essential process.

A static analysis feature U_1, a dynamic analysis feature U_2, and a symbolic execution feature U_3 are connected to get an integration characteristics as shown below:

$$Q^i = f_{concat}(U_1^i, U_2^i, U_3^i) \tag{20}$$

Figure 8. The process of integrating different characteristics



Finally, Q^i is fed into a four-layer perceptron network for learning. To better understand and capture the interactions between modal features, the authors use multiple fully-connected layers, including FC_2, FC_3, and FC_4, stack them, and then perform deep fusion by convolutional operations and maximum pooling operations. The stacked layer is a 512×3 matrix that is convolved by several $1 \times 1 \times 3$ convolution kernels and then fused to get the deeply merged characteristics. In the network, three stacking layers share weights. The convolution operation and maximum pooling operation can be represented in the following form:

$$F^i = f_{pooling}\left(f_{conv}(F_2^i, F_3^I, F_4^i)^{\theta^{conv}}\right) \tag{21}$$

Finally, the fused features will be fed as input to the Softmax classifier for classification to get the final labels:

$$L^i = f_{soft\max}(F^i; \theta^{soft\max}), L^i \in R^C \tag{22}$$

where θ^{conv} and $\theta^{soft\max}$ represent the parameters of the convolutional and soft layers. Considering that the authors used the cross-entropy loss function in this paper, it is possible to represent the model loss function in the following form:

Algorithm 3. Implementation process

Input: The fused feature vectors are fed into the PSO algorithm.
Step 1: Initializing the particle swarm.
A set of particles is initialized based on the dimensionality of the fused feature vectors and the problem size. Each particle represents a potential solution, and the position and velocity of the particle are randomly generated.
Step 2: Compute the fitness function.
Define a fitness function to evaluate the strengths and weaknesses of each particle. This fitness function can be a statistical or machine learning model based on known vulnerability samples or some heuristic rules.
Step 3: Update particle velocity and position.
Based on the results of the evaluation of the fitting function, each particle adjusts its velocity and position according to its optimal solution and global optimal solution. The specific update steps can be implemented according to the basic principles of the algorithm.
Step4: Update individual optimal solution and global optimal solution.
After each update, we need to refresh the individual optimal solution and the global optimal solution of each particle. The optimal solution of each particle is the best in its history, while the global optimal solution is the best in the history of the whole particle swarm.
Step 5: Judge the end condition.
Iteratively update the particle swarm until the preset end condition is satisfied, such as reaching Maximum Iterations or the value of the fitness function converges to a certain threshold.
Output: Output the feature vector corresponding to the global optimal solution, which may represent a potential vulnerability or a further confirmation of a previously detected vulnerability.

$$J\left(\theta^{conv},\theta^{soft\max}\right)=\sum_{i=0}^{u-1}-\log(L^{i})=f_{soft\max}(f_{pooling}(f_{conv}(\boldsymbol{f}_{multi-layer}(U_{1}\mid U_{2}));\theta^{conv})\theta^{soft\max}) \tag{23}$$

where $f_{multi-layer}$ each is a multilayer perceptron manipulation and "|" denotes a connection manipulation.

Detection of Fused Feature Vectors Using Particle Swarm Optimization Algorithm

In their previous work, the authors extracted features from smart contracts from multiple perspectives, including static analysis, dynamic analysis, and symbolic execution, respectively, and obtained multifaceted feature vectors. These features cover the structural, behavioral, and execution characteristics of the contract. Next, they fuseed these features to form a new set of comprehensive feature vectors.

The resulting set of comprehensive feature vectors can comprehensively describe the characteristics of smart contracts, including their structural, behavioral, and execution characteristics. This enables the authors to more accurately understand the potential vulnerabilities of a contract or further confirm previously detected vulnerabilities.

In order to discover more potential vulnerabilities or further confirm previous vulnerabilities, the researchers feed this comprehensive set of feature vectors into a PSO algorithm. This algorithm efficiently searches and identifies the feature vectors that best describe the potential vulnerabilities.

Through this processing flow, it is possible to improve the vulnerability detection accuracy and coverage of smart contracts, thus better guaranteeing the security and reliability of the contracts. The particular implementation process is shown in Algorithm 3.

RESULTS AND DISCUSSION

In the above work, the authors use multimodal feature fusion techniques to continue detecting vulnerabilities in smart contracts. Precisely, they fused features extracted from multiple aspects, such as static analysis, dynamic analysis and symbolic execution, to generate a comprehensive set of

Table 4. Contract Vulnerability Detection Results

	IntegerOverflow	OutOfGas	Tx.Origin	Reentrancy
Detection mode	SIO003	sog001	sto002	ree001
Risk level	3	1	3	3
Language	Solidity	JAVA	Solidity	Solidity
Number of rows of error codes	7 rows 10 columns	6 rows 8 columns	21 rows 8 columns	6 rows 31 columns
Error code content	balances[msg.sender]-amount>0	for(uint256i=0;i<b;i++) {a=b/c*2;}	require(tx.origin==owner)	call.value(amount)("test")

feature vectors. The authors then feed this set of feature vectors into the PSO algorithm to discover more potential vulnerabilities and further confirm the vulnerabilities previously detected using the PSO algorithm.

To validate the effectiveness of this approach, the authors chose to analyze smart contract code that is susceptible to common smart contract vulnerabilities, including integer overflows, reentrant attacks, incorrect random number generation, and denial of service attacks. These code snippets are intended to test whether our approach can correctly detect the specifics of these known vulnerabilities, including the vulnerability's risk level, the language used, and the location in the code where it already appears explicitly (Table 4).

Regarding scalability, the proposed algorithm adopts a PSO algorithm and multimodal feature fusion technique, both of which have good scalability. The PSO algorithm can efficiently handle large-scale optimization problems by simulating the search behavior of particles in the solution space. The multimodal feature fusion technique, on the other hand, can fuse the feature information of different modalities to improve the generalization ability of the algorithm. Therefore, when facing a large number of smart contract datasets, the proposed algorithm can adapt to different sizes of datasets by adjusting the number of particles and the search strategy to achieve efficient processing.

Regarding efficiency, the proposed algorithm is designed to reduce computational complexity and increase processing speed. By processing the byte code of smart contracts through a graph embedding network, the complex contract information can be transformed into a low-dimensional vector representation, thus simplifying the subsequent processing. Meanwhile, the fast convergence property of the PSO algorithm also helps to improve the efficiency of the algorithm. In practical applications, the processing speed of the algorithm can be further improved by optimizing the algorithm parameters and adopting parallel computing.

Experimental Evaluation

The authors implemented contract preprocessing and slicing in the Java environment, graph embedding network, and hazardous contract detection in Python. The experimental setup has 8 GB RAM, four 3.20 GHz cores, and a 2 TB hard disk. Table 5 shows the detailed configuration of the experimental environment.

The authors' experiments are based on the following three datasets:

Dataset I: No Source Code Contracts. This dataset contains more than 2 million bytecode-only smart contracts. The authors use Mist to retrieve blockchain data and get all the bytecodes of the contracts deployed on the blockchain.

Dataset II: Open Source Contracts. Although the authors only considered the bytecode of the contract during our testing, having access to the source code of an open-source contract can

Table 5. Experimental Environment Information Sheet

Software	Random access memory (RAM)	8 GB
Hardware	CPU	Intel(R) Core(TM) i7-11800H @ 3.20GHz
	Systems	Window10
	Java environment	JDK1.82
	Python	Python3.9.6
	Editor (software)	IntelliJ IDEA
	ANTLR	Antlr4.9.6

significantly improve the credibility of the audit results. Since some contract authors have validated and made the source code of their contracts publicly available on Etherscan and the Etherscan browser, the contracts in this dataset have been validated and made public. The contracts in this dataset were collected before the Etherscan policy was adjusted. Since some contracts were validated and made public multiple times, we removed duplicate contracts from the dataset. In the end, the dataset contains 32,499 open-source smart contracts.

Dataset III: Vulnerable Contracts. The contracts in this dataset are used as seeds for matching. The authors retrieved many vulnerable contracts by searching CVEs (Huang et al., 2021) and selected a representative set of contracts that could cause actual harm. Finally, the dataset contains 25 contracts, of which 11 are related to integer overflow vulnerabilities, five are related to reentry vulnerabilities, four are related to nasty random source vulnerabilities, three are related to access control vulnerabilities, and two are related to missed exception handling vulnerabilities. To make the experimental results more valid, the authors removed all contracts that appear in Dataset III from the first two datasets.

Next, they evaluated the efficiency of the experiment. The process took about 57 hours. Figure 9 shows the cumulative distribution of the slice lengths, and Figure 10 shows the distribution of the number of times some of the instructions.

They found that the length of most slices is less than 10, and many instructions appear more than once, indicating that simple cut instructions must be more representative.

The two phenomena indicate that the instructions alone in the slices are not representative enough; therefore, in normalization, this article distinguished the instructions by parameters with

Figure 9. Statistics About the Length of Slices

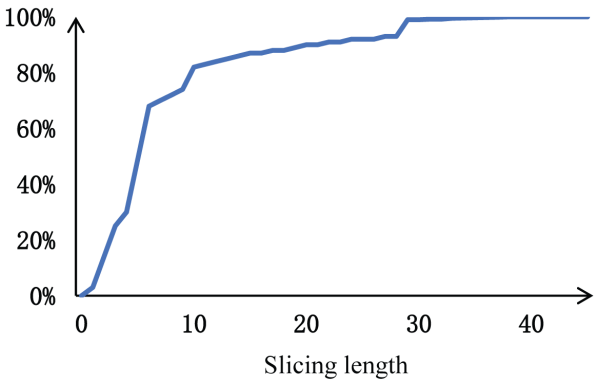
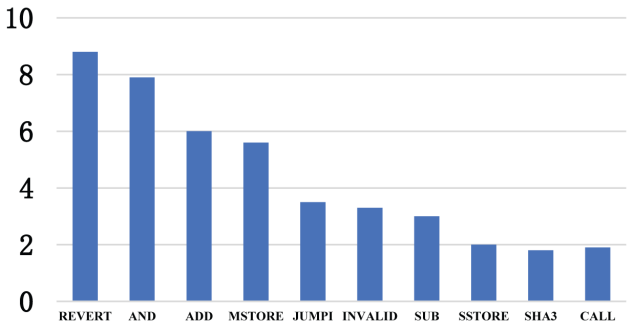


Figure 10. The Usage Time of Instructions



attribute labels. Therefore, they distinguished the instructions by parameters with attribute labels in the normalization process. They also sliced and normalized the contracts' bytecodes in Dataset II, which took about 36 minutes. The slices obtained have similar statistical distributions as in Dataset I. After that, the authors sliced and normalized the contracts in Dataset II using a graph embedding network. Then, they embedded the normalized slices into the vector space using a graph embedding network, which took about 200 minutes.

The authors further sliced Dataset III and selected the slices representing the vulnerabilities, which they chose as the seeds for matching to build the vulnerability vector library. The researchers then computed the similarity between the sliced vectors of Dataset I and Dataset II. The average detection time per known vulnerability contract is about 18 minutes.

Assessment of Indicators

The authors applied four standard evaluation criteria to assess the effectiveness of their methods and compare them to other methods. Accuracy (ACC) is the percentage of all samples correctly identified and is used as an overall measure of detection capability. Precision (P) is the percentage of identified vulnerable samples out of all identified attack samples and reflects the accuracy of the detection results. Recall (R), on the other hand, reveals the percentage of identified vulnerable samples to all actual attack samples, reflecting the comprehensiveness of the method. In addition, the F1 score (F1) is the average of precision and recall, providing us with a comprehensive evaluation perspective. Also, the authors utilized the AUC value, which is the area under the region of convergence (ROC) curve, to further compare the performance of different classifiers. The exact formula for the evaluation metrics is as follows:

$$ACC = \frac{TP + TN}{TP + FP + TN + FN} \tag{24}$$

$$R = \frac{TP}{TP + FN} \tag{25}$$

$$P = \frac{TP}{TP + FP} \tag{26}$$

$$F_1 = 2 \times \frac{P \times R}{P + R} \tag{27}$$

Analysis and Comparison of Experimental Results

Using this approach, the authors randomly selected 3000 contracts from Dataset II for vulnerability detection and calculated the false alarm rate and false judgment rate under different threshold conditions. By adjusting the balance between these two metrics, the similarity threshold was finally set to 90%. In addition, in order to balance the efficiency and effectiveness, the authors also adjusted the embedding size S in the graph embedding network, and the final value of S was 64. These parameters can be adjusted as needed for future use, and the settings here can be used as a valuable reference. The dataset employs a smart contract vulnerability detection method based on bytecode matching and PSO algorithm I (Ding et al., 2019). A total of 1,220 vulnerable contracts were reported, of which the number of integer overflow, reentrant, erroneous random source, access control, and missing exception handling vulnerabilities were 732, 129, 23, 182, and 154, respectively. To audit these reactive contracts, the authors deployed them on a private chain and attempted to exploit the vulnerabilities in them. In practice, manually inspecting the underlying bytecode is very time-consuming and unreliable, as it requires to be able to infer the behavior in the source code from the bytecode, deduce the location of the vulnerabilities and the triggering method, and then test for the existence of the vulnerabilities. Therefore, the authors only manually inspected the top 10 contracts with the highest similarity to known vulnerable contracts and successfully triggered vulnerabilities in 8 contracts. In Dataset II, they found 152 contracts with exploitable vulnerabilities, corresponding to 4, 23, 10, 11, and 5 out of 104. The results also contained 19 false positives. As a result, the accuracy of the method on Dataset II is as high as 88.89%.

In order to evaluate the performance of the PSO algorithm and multimodal feature fusion technique for smart contract vulnerability detection, the authors used the same Dataset II and compared it with existing smart contract vulnerability detection tools, including Mythril (Mythril, 2023), Smartcheck (Zhuang et al., 2020), Slither (Zhou et al., 2021), as well as TMP (Huang et al., 2021), which the authors chose as a similar comparison method. Table 6 lists the detailed differences between these approaches and our proposed approach in various aspects.

The authors conducted comparative experiments using various smart contract vulnerability detection tools such as Mythril, Smartcheck, Slither, and TMP under different evaluation metrics (i.e., ACC, AUC, P, R, and F1). The experimental Dataset II is the same as the method the authors described in this paper, and Tables 7 and 8 show the experimental results. Among them, Mythril is a well-known security analysis tool provided by the Ethernet open source community, which can deeply analyze the security vulnerabilities in Solidity smart contracts; Smartcheck is an extensible static analysis tool for detecting vulnerabilities or code problems in smart contracts; Slither is an open-source static analysis framework for Solidity, which can quickly locate the vulnerabilities; TMP

Table 6. Comparison of Existing Methods and the Authors' Proposed Method

Method	Year	Technical category	Vulnerability types
Mythril	2017	Symbolic execution	10%
Smartcheck	2018	Static analysis	30%
Slither	2019	Static analysis	35%
TMP	2020	Deep learning	80%
Our me thod	_	Deep learning	95%

(Chen et al., 2021) is a novel static analysis tool for detecting vulnerabilities in smart contracts. Also, it is also a novel temporal information dissemination network that uses graph convolution to learn vulnerability features in normalized contract graphs (Yang et al., 2024).

Experimental results show that the method proposed in this paper can support more types of vulnerability detection without relying on expert knowledge. Compared with existing smart contract vulnerability detection tools, the method performs better in terms of detection accuracy and AUC value for various types of vulnerabilities, and the overall detection performance is superior. Figure 11 shows the specific performance.

Traditional smart contract vulnerability detection methods rely too much on experts' experience and constantly need to update the feature library. TMP, as a vulnerability detection method based on the concept of contract graph, has a unique advantage in detecting security vulnerabilities caused by intercontract calls and cannot be applied to detect vulnerabilities in one line of code, such as ARTHM. Therefore, in this paper the authors propose a smart contract vulnerability detection method based on PSO algorithm and multimodal feature fusion technology, which makes use of as much contract feature information as possible and organically combines this information, thus outperforming existing methods in vulnerability detection.

The use of an approach based on PSO algorithm and multimodal feature fusion techniques has been shown to have significant advantages in the task of detecting four common smart contract vulnerabilities. Compared to traditional detection methods, this approach excels in improving both the accuracy of the model (ACC value) and the area under the curve (AUC value) of the detection performance (Yang et al., 2021). Using our proposed new approach, we are able to detect vulnerabilities in smart contracts more accurately and efficiently.

In-Depth Statistical Analysis

In order to verify the effectiveness of the smart contract vulnerability detection scheme proposed in this paper, the authors conducted an in-depth statistical analysis. First, they collected a large number of smart contract samples, including contracts with known vulnerabilities and standard contracts. Then, they utilized the proposed scheme to detect these contracts and count the metrics such as the accuracy rate, false alarm rate, and missed alarm rate of the detection results. In addition, they analyzed the impact of different parameter settings on the detection results to determine the optimal parameter configuration.

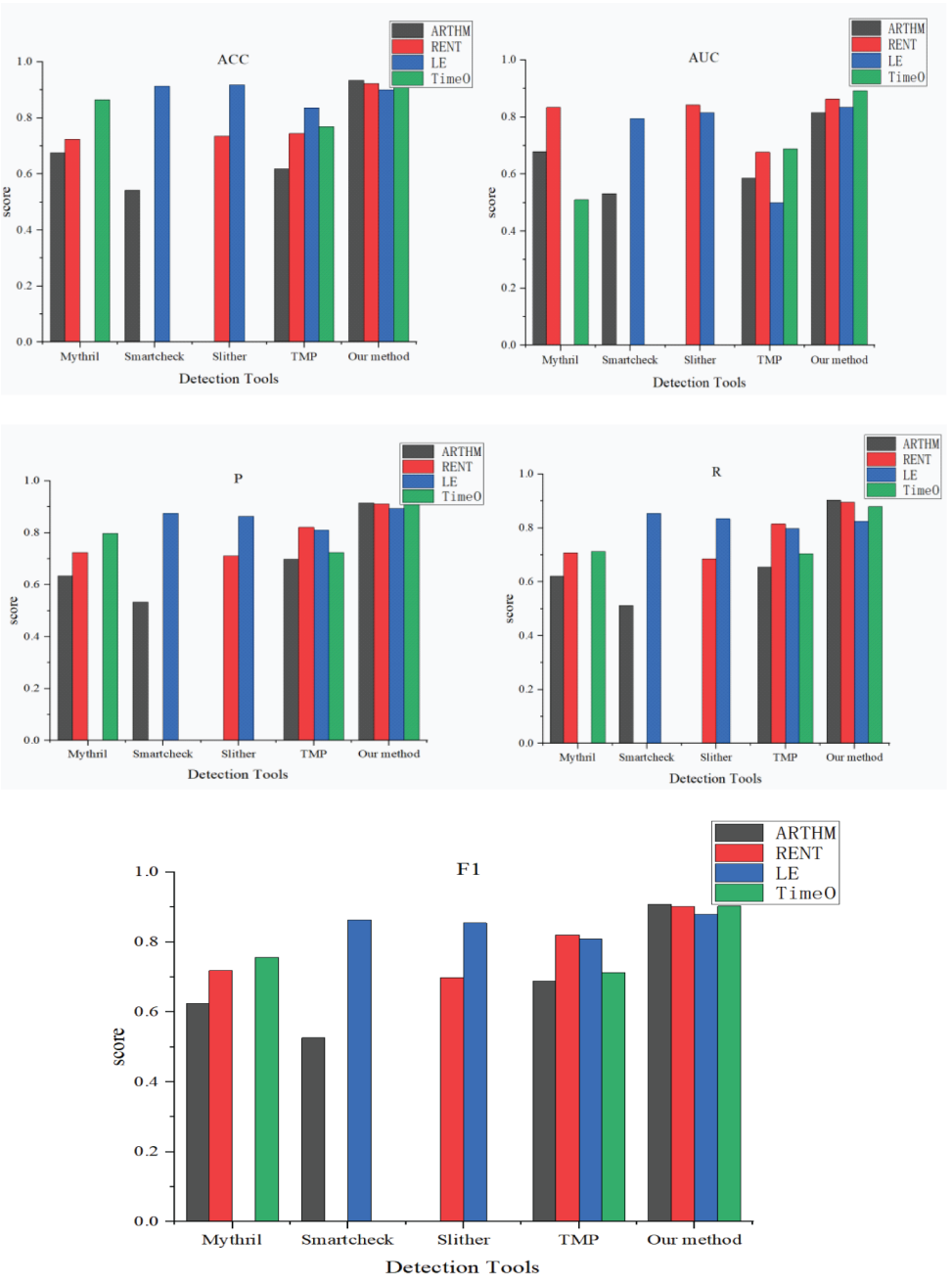
Comparison With Benchmark Methods

In order to evaluate the smart contract vulnerability detection scheme proposed in this paper more comprehensively, the authors also compared it with existing benchmark methods. These benchmark methods include traditional rule-based detection methods. The authors tested these methods using the same smart contract sample set and compared their detection performance. The experimental results show that the scheme proposed in this paper outperforms the benchmark methods in terms of accuracy, false positives and misses, especially in the detection of complex and hidden vulnerabilities.

CONCLUSION

In this paper, the authors propose an innovative smart contract vulnerability detection method that combines PSO algorithm and multimodal feature fusion technique. By applying graph embedding network to deeply analyze the byte code of smart contracts, it is possible to effectively capture the structural and behavioral information of the contracts. Meanwhile, the introduction of PSO algorithm makes the whole detection process more efficient and enables the optimized search of parameters. In addition, the authors also utilize the multimodal feature fusion technique to comprehensively consider various contract feature information, thus further improving the accuracy and efficiency of vulnerability detection. Experimental results show that the method proposed in this paper exhibits

Figure 11. Performance Evaluation of Different Types of Vulnerability Detection Methods Using Different Evaluation Metrics



higher detection accuracy and AUC value compared with existing detection tools. Further, the method does not completely rely on expert knowledge, and thus can support more types of vulnerability detection, greatly broadening its application scope.

However, it is crucial to also face up to the limitations and challenges of existing methods. On the one hand, these methods have a high dependence on the quality and quantity of data,

Table 7. Experimental Comparison With Other Methods

Method	ARTHM		RENT		LE		TimeO	
	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC
Mythril	0.675	0.679	0.724	0.834	_	_	0.865	0.51
Smartcheck	0.542	0.531	_	_	0.913	0.795	_	_
Slither	_	_	0.735	0.842	0.918	0.815	_	_
TMP	0.618	0.586	0.744	0.677	0.836	0.499	0.768	0.689
The authors' method	0.934	0.815	0.923	0.863	0.901	0.835	0.951	0.892

Table 8. Experimental Comparison With Other Methods

Method	ARTHM			RENT			LE			TimeO		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Mythril	0.634	0.621	0.625	0.724	0.708	0.719	_	_	_	0.798	0.713	0.756
Smartcheck	0.533	0.512	0.526	_	_	_	0.875	0.854	0.864	_	_	_
Slither	_	_	_	0.711	0.685	0.698	0.863	0.834	0.855	_	_	_
TMP	0.698	0.655	0.688	0.821	0.815	0.820	0.811	0.798	0.809	0.724	0.705	0.713
The authors' method	0.915	0.904	0.909	0.912	0.896	0.902	0.893	0.824	0.879	0.923	0.879	0.903

which limits their effectiveness in practical applications to some extent. On the other hand, the complexity of the algorithms and the resource consumption problem when dealing with large-scale datasets are also challenges that cannot be ignored. In addition, when facing new or unknown vulnerabilities, the existing models may seem overwhelming and need to be constantly updated and optimized.

Therefore, future research should focus on solving these problems. First, it is necessary to improve the quality and diversity of data to ensure that the model can identify various vulnerabilities more accurately. Second, it is also crucial to optimize the performance and efficiency of the algorithms, which will help improve the speed and accuracy of detection. In addition, realizing adaptive and online learning capabilities, as well as exploring vulnerability detection techniques under cross-chain security and privacy protection, are also important future research directions.

At the same time, it is also necessary to pay attention to the ethical considerations of vulnerability detection techniques. When applying these technologies, scholars must respect the privacy and copyright of contract developers and ensure the legality and compliance of the whole detection process. It is also crucial to minimize false positives and omissions to avoid unnecessary losses and crises of trust for developers and users. Finally, it is necessary to make it clear that these technologies are not all-purpose security measures, but should work together with other security measures to form a multilayered security protection system.

In summary, although smart contract vulnerability detection technology has a wide range of application prospects and research value, researchers still need to continuously optimize and improve it to give full play to its role in smart contract security.

AUTHOR CONTRIBUTIONS

T.F. participated in the feasibility discussion, analysis of the paper scheme, and the proofreading of the paper; Y.C. was responsible for the overall design, performance analysis, and paper writing. All the authors have read and agreed to the published version of the manuscript.

COMPETING INTERESTS

No competing interests exist.

FUNDING

This work is supported by the National Natural Science Foundation of China (Grant No. 62162039, 61762060).

REFERENCES

Alweshah, M., Khalaileh, S. A., Gupta, B. B., Almomani, A., Hammouri, A. I., & Al-Betar, M. A. (2020). The monarch butterfly optimization algorithm for solving feature selection problems. *Neural Computing & Applications*, *34*, 1–15.

Chawra, V. K., & Gupta, G. P. (2022). Optimization of the wake-up scheduling using a hybrid of memetic and tabu search algorithms for 3D-wireless sensor networks. *International Journal of Software Science and Computational Intelligence*, *14*(1), 1–18. doi:10.4018/IJSSCI.300359

Chen, X., Liao, P., Zhang, Y., Huang, Y., & Zheng, Z. (2021). Understanding code reuse in smart contracts. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2021)* (pp. 470—479). IEEE. doi:10.1109/SANER50967.2021.00050

Datar, M., Altman, E., De Pellegrini, F., El Azouzi, R., & Touati, C. (2020). A mechanism for price differentiation and slicing in wireless networks. In Proceedings of the 2020 18th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT) (pp. 1–8). IEEE.

Ding, S. H. H., Fung, B. C. M., & Charland, P. (2019). Asm2 Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings for the IEEE Symposium on Security and Privacy SP*) (pp. 472–489).

Fatemidokht, H., Rafsanjani, M. K., Gupta, B. B., & Hsu, C. H. (2021). Efficient and secure routing protocol based on artificial intelligence algorithms with UAV-assisted for vehicular ad hoc networks in intelligent transportation systems. *IEEE Transactions on Intelligent Transportation Systems*, *22*(7), 4757–4769. doi:10.1109/TITS.2020.3041746

Hamp-Lyons, L. (1990). Second Language Writing: Assessment Issues. In B. Kroll (Ed.), *Second writing: Research insights for classroom* (pp. 69–87). Cambridge University Press. doi:10.1017/CBO9781139524551.009

Hu, B., Gaurav, A., Choi, C., & Almomani, A. (2022). Evaluation and comparative analysis of semantic web-based strategies for enhancing educational system development. *International Journal on Semantic Web and Information Systems*, *18*(1), 1–14. doi:10.4018/IJSWIS.302895

Kumar, A., & Sivakumar, P. (2022). Cat-squirrel optimization algorithm for VM migration in a cloud computing platform. *International Journal on Semantic Web and Information Systems*, *18*(1), 1–23.

Liu, Y., Liu, L., Guo, Y., & Lew, M. S. (2018). Learning visual and textual representations for multimodal matching and classification. *Pattern Recognition*, *84*, 51–67. doi:10.1016/j.patcog.2018.07.001

Lu, J., Shen, J., Vijayakumar, P., & Gupta, B. B. (2021). Blockchain-based secure data storage protocol for sensors in the industrial internet of things. *IEEE Transactions on Industrial Informatics*, *18*(8), 5422–5431. doi:10.1109/TII.2021.3112601

Madan, K., & Bhatia, R. K. (2021). Ranked deep web page detection using reinforcement learning and query optimization. *International Journal on Semantic Web and Information Systems*, *17*(4), 99–121. doi:10.4018/IJSWIS.2021100106

Madhumala, R. B., & Tiwari, H. (2022). Resource optimization in cloud data centers using particle swarm optimization. *International Journal of Cloud Applications and Computing*, *12*(2), 1–12. doi:10.4018/IJCAC.305856

Metz, C. (2021, January 22). *The biggest crowdfunding project ever-the DAO-is kind of a mess*. https://www.wired.com/2016/06/biggest-crowdfunding-project-ever-dao-mess

Momeni, P., Wang, Y., & Samavi, R. (2019). Machine learning model for smart contracts security analysis. In *Proceedings of the 17th International Conference on Privacy, Security and* Trust (pp. 1—6). IEEE. doi:10.1109/PST47121.2019.8949045

Mythril. (2023). [Data set]. https://github.com/ConsenSys/mythril

Nedjah, N., Cardoso, A. V., Tavares, Y. M., Mourelle, L. D. M., Gupta, B. B., & Arya, V. (2023). Co-design dedicated system for efficient object tracking using swarm intelligence-oriented search strategies. *Sensors (Basel)*, *23*(13), 5881. doi:10.3390/s23135881 PMID:37447729

Nguyen, D. T., Pham, L. H., Sun, J., Lin, Y., & Tran, M. Q. (2020). sFuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the 42nd International Conference on Software Engineering* (pp. 778-788). IEEE Computer Society. doi:10.1145/3377811.3380334

Nguyen, G. N., Le Viet, N. H., Elhoseny, M., Shankar, K., Gupta, B. B., & Abd El-Latif, A. A. (2021). Secure blockchain enabled Cyber–physical systems in healthcare using deep belief network with ResNet model. *Journal of Parallel and Distributed Computing*, *153*, 150–160. doi:10.1016/j.jpdc.2021.03.011

Nhi, N. T. U., & Le, T. M.Thanh The Van. (2022). A model of semantic-based image retrieval using C-tree and neighbor graph. *International Journal on Semantic Web and Information Systems*, *18*(1), 1–23. doi:10.4018/IJSWIS.295551

Ni, Y. D., Zhang, C., & Yin, T. T. (2020). A survey of smart contract vulnerability research. *Journal of Cybersecurity*, *5*(3), 78–99.

Parizi, R. M., Dehghantanha, A., Choo, K. K. R., & Singh, A. (2018). Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering* (pp. 103—113). IBM Corp.

Raj, M. G., & Pani, S. K. (2022). Chaotic whale crow optimization algorithm for secure routing in the IoT environment. *International Journal on Semantic Web and Information Systems*, *18*(1), 1–25. doi:10.4018/IJSWIS.300824

SafeMath. (2022). https://docs.statechannels.org/contractapi/natspec/SafeMath

Sayeed, S., Marco-Gisbert, H., & Caira, T. (2020). Smart contract: Attacks and protections. *IEEE Access : Practical Innovations, Open Solutions*, *8*, 24416–24427. doi:10.1109/ACCESS.2020.2970495

Singh, S., Kumar, R., & Rao, U. P. (2022). Multi-objective adaptive manta-ray foraging optimization for workflow scheduling with selected virtual machines using time-series-based prediction. *International Journal of Software Science and Computational Intelligence*, *14*(1), 1–25. doi:10.4018/IJSSCI.312559

Sissodia, R., Rauthan, M. S., & Barthwal, V. (2022). A multi-objective optimization scheduling method based on the genetic algorithm in cloud computing. *International Journal of Cloud Applications and Computing*, *12*(1), 1–21. doi:10.4018/IJCAC.305217

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018). SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018* (pp. 9–16). ACM. doi:10.1145/3194113.3194115

Wang, H., Li, Y., Lin, S., Ma, L., & Liu, Y. (2019). VULTRON: Catching vulnerable smart contracts once and for all. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (pp. 1—4). IEEE/ACM. doi:10.1109/ICSE-NIER.2019.00009

Wu, H., Zhang, Z., Wang, S., Lei, Y., Lin, B., Qin, Y., Zhang, H., & Mao, X. (2021). Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *Proceedings of the 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)* (pp. 378–389). IEEE. doi:10.1109/ISSRE52982.2021.00047

Xing, C., Chen, Z., Chen, L., Guo, X., Zheng, Z., & Li, J. (2020). A new scheme of vulnerability analysis in smart contract with machine learning. *Wireless Networks*, *2*, 1–10. doi:10.1007/s11276-020-02379-z

Xu, M. X., Yuan, C., Wang, Y. J., Fu, J. H., & Li, B. (2019). Mimic blockchain—Solution to the security of blockchain. *Journal of Software*, *30*(6), 1681–1691.

Yamaguchi, F., Lindner, F., & Rieck, K. (2011). Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies* (pp. 456—462). USENIX Association.

Yamaguchi, F., Lottmann, M., & Rieck, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*(pp. 3–7). ACM. doi:10.1145/2420950.2421003

Yang, H., Gu, X., Chen, X., Zheng, L., & Cui, Z. (2024). CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection. *Science of Computer Programming*, *234*, 103076. doi:10.1016/j.scico.2023.103076

Yang, Y. (1999). An evaluation of statistical approaches retrieval to text categorization. *Information (Basel)*, *1*(1), 76–88.

Yang, Z., Keung, J., Yu, X., Gu, X., Wei, Z., Ma, X., & Zhang, M. (2021). A multi-modal transformer- based code summarization approach for smart contracts. In *Proceedings of 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC 2021)* (pp. 1–12). IEEE. doi:10.1109/ICPC52881.2021.00010

Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., & Oliva, A. (2014). Learning deep features for scene recognition using places database. *Advances in Neural Information Processing Systems*, *27*, 426–435.

Zhou, L., Qin, K., Cully, A., Livshits, B., & Gervais, A. (2021). On the just-in-time discovery of profit-generating transactions in DeFi protocols. In *Proceeding of the 42nd IEEE Symposium on Security and Privacy (SP 2021)* (pp. 919-936). IEEE. doi:10.1109/SP40001.2021.00113

Zhou, Z., Wang, B., Gu, B., Ai, B., Mumtaz, S., Rodriguez, J., & Guizani, M. (2020). Time-dependent pricing for bandwidth slicing under information asymmetry and price discrimination. *IEEE Transactions on Communications*, *68*(11), 6975–6989. doi:10.1109/TCOMM.2020.3001050

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., & He, Q. (2020). Smart contract vulnerability detection using graph neural network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence—IJCAI 2020* (pp. 3283–3290). ACM. doi:10.24963/ijcai.2020/454

Feng Tao (1970), Male (Han), Lanzhou, Gansu Province, professor, Ph.D., Research Area: Network and Information Security, Blockchain Security, Smart Contract Security.

Cui Yuyang (1998), Male (Han), Yingkou, Liaoning Province, China, Position: Master's Degree, Education: Master's Degree, Research Field: Smart Contract Security.