The Journal of Systems & Software 195 (2023) 111550

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Combine sliced joint graph with graph neural networks for smart contract vulnerability detection☆

Jie Cai [a], Bin Li [a,*], Jiale Zhang [a,*], Xiaobing Sun [a], Bing Chen [b]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

ARTICLE INFO

Article history:
Received 20 March 2022
Received in revised form 6 July 2022
Accepted 14 October 2022
Available online 20 October 2022

Keywords:
Smart contract
Vulnerability detection
Code representation
Graph neural network

ABSTRACT

Smart contract security has drawn extensive attention in recent years because of the enormous economic losses caused by vulnerabilities. Even worse, fixing bugs in a deployed smart contract is difficult, so developers must detect security vulnerabilities in a smart contract before deployment. Existing smart contract vulnerability detection efforts heavily rely on fixed rules defined by experts, which are inefficient and inflexible.

To overcome the limitations of existing vulnerability detection approaches, we propose a GNN based approach for smart contract vulnerability detection. First, we construct a graph representation for a smart contract function with syntactic and semantic features by combining abstract syntax tree (AST), control flow graph (CFG), and program dependency graph (PDG). To further strengthen the presentation ability of our approach, we perform program slicing to normalize the graph and eliminate the redundant information unrelated to vulnerabilities. Then, we use a Bidirectional Gated Graph Neural-Network model with hybrid attention pooling to identify potential vulnerabilities in smart contract functions. Empirical results show that our approach can achieve 89.2% precision and 92.9% recall in smart contract vulnerability detection on our dataset and reveal the effectiveness and efficiency of our approach.

1. Introduction

Blockchain is a distributed ledger proposed in 2008 which provides a reliable decentralized execution model (Nakamoto, 2008). The smart contract is the most critical part of the blockchain as an executable code developed by the high-level programming language (e.g. Solidity Wood et al., 2014a) and running on blockchain (Szabo, 1997). The decentralization and trustworthy properties of the blockchain-based smart contract have attracted considerable attention from different industries. Such as in IoT, many works try to apply smart contracts to do access control, task management or data management (Zhang et al., 2018; Park et al., 2018; Hang and Kim, 2020). Until now, in the most famous blockchain platform, Ethereum (Wood et al., 2014b), there are already up to 1.5 million smart contracts running on it (Pierro et al., 2020).

However, similar to other software programs, smart contracts may also have vulnerabilities. Even worse, some properties of the smart contract make itself more attractive to hackers. First, as a smart contract always manages cryptocurrency or sensitive data, its vulnerabilities will cause substantial losses. Second, smart contracts deployed on the blockchain are transparent and exposed to all blockchain users, allowing hackers to call a smart contract with no limitations. Finally, as the blockchain is immutable and irreversible when developers deploy a vulnerable smart contract on the blockchain, the developers neither can fix the contract's code nor interrupt the execution of the contract. So far, there have been several serious accidents (Falkon, 2017; Parity Technologies, 2017; Wright, 2021; Osborne, 2021) caused by vulnerabilities in the smart contract, the most influential one is TheDao accident (Falkon, 2017) caused by the reentrancy vulnerability, and about $70 million was stolen during this accident in 2016. Therefore, detecting smart contract vulnerabilities has become a vital and immense challenge to solve urgently.

Most of the current smart contract vulnerability detection approaches (Luu et al., 2016a; Tsankov et al., 2018a; Alt and Reitwiessner, 2018; Feist et al., 2019) use static analysis to search vulnerability patterns pre-defined by experts in target smart contracts. These manually defined vulnerability patterns are relatively simple and cannot cover some complex vulnerability patterns, which will cause a high rate of false negatives. Moreover, with the rapid growth of smart contracts, it has become

☆ Editor: Earl Barr.

* Corresponding authors.
E-mail addresses: jiecaiyzu@gmail.com (J. Cai), lb@yzu.edu.cn (B. Li), jialezhang@yzu.edu.cn (J. Zhang), xbsun@yzu.edu.cn (X. Sun), cb_china@nuaa.edu.cn (B. Chen).

https://doi.org/10.1016/j.jss.2022.111550
0164-1212/

challenging to design precise vulnerability patterns through a few experts.

Researchers have proposed various machine learning-based approaches for smart contract vulnerability detection to avoid relying on expert-defined patterns. Some works (Tann et al., 2018; Qian et al., 2020) treat the smart contract as natural language and represent it as a flat sequence, then use a sequential neural network (e.g., LSTM or RNN) to conduct feature learning and vulnerability detection. However, a smart contract has more complex logic and structure information inside than natural languages. These approaches will lose the subtle semantic features such as data or control dependencies within the code's structure and give poor accuracy in the vulnerability detection task.

To preserve these non-sequential semantic features of the smart contract code, some graph-based approaches are proposed (Zhuang et al., 2020; Qian et al., 2020). These works propose to convert the source code of a contract into a graph and use graph neural networks to detect vulnerabilities. However, there are some limitations. First, their detection granularity is contract-level which is too coarse. There are always several functions in a contract, and contract-level vulnerability detection is not convenient for developers. Second, their code graph representation cannot comprehensively reflect a smart contract's vulnerability related semantic feature. Most of them consider part of smart contract's features, as in Zhuang et al. (2020) use part of CFG to represent the entire contract's semantic feature without syntax or dependency features. Moreover, the graph representation may contain some vulnerable irrelevant components which may interfere model's feature learning. Finally, it uses a unidirectional graph convolution network to learn the features of a graph, which constructs the node feature along each node's outgoing edges, and this restricts the model from learning each node's contextual feature from incoming edges.

As the Solidity (Wood et al., 2014a) is the dominating programming language to implement the smart contract, most researches (Luu et al., 2016a; Tsankov et al., 2018a; Alt and Reitwiessner, 2018; Feist et al., 2019; Tann et al., 2018; Qian et al., 2020) on vulnerability detection for smart contracts select the smart contracts written by Solidity as the research object. To facilitate comparison and analysis, we also select the smart contracts written by Solidity as the research object to do the vulnerability detection at the function-level in our approach.

To cope with the above challenges, we propose a graph neural network-based approach to detect smart contract vulnerabilities at slice-level. First, to reflect sufficient semantic features from code with less vulnerable irrelevant information, we represent the smart contract function as a sliced joint code graph by combining different graph representations (e.g., AST or CFG). Each graph representation can reflect different syntax or semantic information of the smart contract function. Combining these different graphs can take advantage of their syntax or semantic representation ability and make our approach more comprehensive. Moreover, the program slicing technique can eliminate vulnerability-irrelevant information as much as possible from this code graph representation to migrate the interference of these irrelevant pieces of information in model training. Second, we use a bidirectional gated graph neural network with the hybrid attention pooling to extract the vulnerability features from our graph representation for vulnerability detection. The bidirectional graph can enhance the model's ability in contextual features learning. Furthermore, the hybrid attention pooling can help the detection model focus on vulnerability features and suppress unimportant ones via the attention mechanism. We evaluated the proposed approach on a dataset with nine common types of vulnerabilities, and the experimental results demonstrate that our approach can detect vulnerabilities more accurately than the baselines.

The main contributions of this paper are as follows:

- We propose a new approach to encode smart contract's function into a graph with sufficient semantic features and few noise nodes which are irrelevant to vulnerability. We employ several slice criteria specific to smart contract vulnerabilities to preserve the vulnerability related source code.
- We propose a bidirectional gated graph neural network with a hybrid attention pooling layer to learn the code features, efficiently capturing vulnerability-related features from the graph for vulnerability detection.
- We perform experiments on a dataset with nine common types of vulnerabilities and demonstrate that our approach can achieve better performance than the state-of-the-art approaches on vulnerability detection.

The rest of the paper is organized as follows. In Section 2, we describe the background of the smart contract and graph neural network. Section 3 explains the motivation of our approach. Section 4 explains the details of our approach. Section 5 shows the experiment setting and implementation of our approach. Section 6 is used to discuss the experimental results. In Section 7, we analyze the threats to validity in the experiments. Section 8 is the related work. Section 9 concludes the paper and discuss the future work.

2. Background

This section briefly introduces background about the smart contract and common vulnerabilities in the smart contract. Then, we briefly introduce the graph neural network.

2.1. Smart contract basics

Smart contracts and Execution environment. A smart contract is a piece of code that runs on the blockchain. After being deployed on the blockchain, it will execute automatically according to the pre-defined code logic. First, developers use high-level language (e.g. solidity) to write smart contracts, compile the smart contracts into bytecode, deploy the compiled bytecode on the blockchain and get a unique 160-bit hexadecimal hash contract address, which is the ID of this smart contract. The user calls the contract code by setting up a transaction request to the contract address, and it automatically executes the contract on the Ethereum virtual machine.

Internal/External function call. Smart contract support two types of function calls. One is named internal function call that calls a function in the same contract. The other is called an external contract's function named external function call. The two types of function calls are quite different during execution in the Ethereum Virtual Machine (EVM). Internal function call only requires instruction jumps during the execution. In contrast, external function calls must send messages to external contracts through the blockchain to trigger this external call operation.

The fallback function. The fallback function is a unique mechanism of smart contracts. It is an anonymous function without input or return value. Each smart contract can only have one fallback function. The fallback function has two usages. The first is that when an external user calls a contract's function, and the function does not exist, the fallback function will execute automatically. The second is that when an external user transfers ether to this contract, the fallback function will execute to receive ether automatically.

2.2. Smart contract vulnerabilities

(1) **Block Information Dependency:** The Ethereum provide some interfaces to facilitate smart contracts to obtain

```
function random(uint max) view private returns (uint256
    result) {
    // Get the best seed for randomness
    uint256 x = salt * 100 / max;
    uint256 y = salt * block.number / (salt / 5);
    uint256 seed = block.number / 3 + (salt / 300) + y;
    uint256 h = uint256(blockhash(seed));
    // Random number between 1 and max
    return uint256((h / x)) / max + 1;
}
```

Fig. 1. Block information dependency vulnerability.

```
function forward(address callee, bytes _data) public {
    callee.delegatecall(_data)
}
```

Fig. 2. Dangerous delegatecall vulnerability.

```
function mintToken(address target, uint256 mintedAmount)
    onlyOwner public {
    balanceOf[target] += mintedAmount;
    totalSupply += mintedAmount;
    Transfer(0, this, mintedAmount);
    Transfer(this, target, mintedAmount);
}
```

Fig. 3. Integer overflow vulnerability.

current block information such as *block.number* or *block.gaslimit*. Some contracts use these block information as a triggering condition to execute some critical operations. However, these block information are fully controlled by miners, which means malicious miners can manipulate a smart contract's behavior by control these block information.

Example: In Fig. 1, the function uses *block.number* as a seed (line 4 and line 5) to generate a pseudo-random number. However, miners can manipulate this *block.number* to control the result of the random number.

(2) **Dangerous Delegatecall:** The *delegatecall* interface allows a smart contract to execute an external contract's code in its own context to facilitate code-reuse. However, if the external contract is a malicious one, it can directly control the caller contract's behavior and modify the state of the caller contract.

Example: In Fig. 2, the function forward uses *delegatecall* to call an external contract at address *callee* (line 2). However, the address is an input parameter of this function make it easy to call an untrusted contract.

(3) **Integer Overflow:** Integer overflow is a common vulnerability. Programming languages have specific length restrictions on the storage space of integer types. So once the result of an integer operation exceeds this range, an integer overflow will occur. Different from traditional software, the integers in smart contracts often represent the number of assets. Once an integer overflow occurs, it will cause substantial economic losses.

Example: In Fig. 3, the function *mintToken* increase the balance of target's account. But the *mintedAmount* can be arbitrary so the add operation (line 2) would result in an integer overflow.

```
function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)());
      credit[msg.sender]-=amount;
    }
}
```

Fig. 4. Re-entrancy vulnerability.

```
function sudicideAnyone() {
    selfdestruct(msg.sender);
}
```

Fig. 5. Suicide contract vulnerability.

(4) **Re-entrancy.** When called a non-recursive function, the execution is atomic in traditional programs, and there will be no new function execution before the current function execution ends. However, in the smart contract, when conducting an external function call, the malicious callee external contract may reenter the caller before the caller contract finishes.

Example: Fig. 4 is an example of re-entrancy vulnerability. It calls the external contract *msg.sender* at line 3 and updates the state variable *credit* of the callee at line 4. This update state variable after external call pattern make the malicious external contract can reenter this function before state update to avoid the check at Line 2.

(5) **Suicide Contract.** Solidity provides the *selfdestruct* interface to destroy the contract, but because of missing or insufficient access controls, malicious parties can destroy the contract.

Example: Fig. 5 is an example of Suicide Contract vulnerability. As shown the *selfdestruct(msg.sender)* has no execution condition, so anyone call destruct this contract.

(6) **Short Address Attack.** When sending a transaction to a smart contract, the user needs to package the transaction data under specification. The smart contract specification requires that the length of the address type data must be 20 bytes. When the address data is less than 20 bytes, it will be completed with subsequent data or 0. When an attacker maliciously packs address data with a length of fewer than 20 bytes, it will cause an error in transaction data parsing. Therefore, developers must strictly check the length of the input data to prevent attackers from attacking the contract.

Example: As shown in Fig. 6, Someone calls this function with the parameter *_to* as 0x123456789012345 67890123456789012345678 and the value of *amount* as 40. However, the length of *_to* is only 19 bytes, which must be 20 bytes as the requirement of the specification, the blockchain platform will utilize the first byte of the second input parameter *amount* to complete the length of *_to* to 20 and pad zero at the end of *amount*. After padding, the value of *_to* will be 0x1234567890123456789012345678901234567800. Moreover, the value of amount will become 200, which is five times the original input value as 40.

(7) **Timestamp dependency.** Some smart contracts use the *block.timestamp* as a constraint of execution paths, and the miner node can adjust the value of the *block.timestamp* within a range of about 900 s. Therefore, the miners who create the block can deliberately choose a timestamp that is beneficial to them.

Example: As shown in Fig. 7, this function generates a hash number by *block.timestamp* as a seed. However, the

```
function sendMoney(address _to, uint256 amount) {
    if (balance[msg.sender] > amount) {
        balance[msg.sender] = balance[msg.sender] - amount;
        balance[_to] = balance[_to] + amount;
    }
}
```

Fig. 6. Short address attack.

```
function() payable {
    if (msg.value >= 10 finney) {
        bytes20 bonusHash = ripemd160(block.timestamp);

        if (bonusHash[0] == 0) {
            uint8 bonusMultiplier = ((bonusHash[1] & 0x01 != 0)
↪ ? 1 : 0);
            uint256 bonusTokensIssued = (msg.value * 100) *
↪ bonusMultiplier;
        }
    }
}
```

Fig. 7. Timestamp dependency vulnerability.

```
function claimReward(uint256 submission) {
    require (!claimed);
    require(submission < 10);

    msg.sender.transfer(reward);
    claimed = true;
}
```

Fig. 8. Transaction order dependency.

block.timestamp can be controlled by the miner so that a malicious miner can control the result of this function.

(8) **Transaction Order Dependency.** Users can send transactions to invocation a smart contract, but the sending time of the transaction and the execution time of the transaction are not strictly related. Miners can adjust the packaging order of transactions according to their own needs to manipulate the order of execution of transactions. If the business logic of the smart contract uses the transaction sequence as the decision condition or the current transaction execution has an impact on the subsequent transaction results, the attacker can attack the contract by manipulating the transaction packaging sequence.
Example: As shown in Fig. 8, this function can send a reward to the first ten users. However, the miner can arrange the transaction execution order. If a malicious miner postpones packaging some user's transaction to this function, this user may lose the chance to get the reward from this function.

(9) **Unchecked Call Return Value.** Some external function call interfaces do not throw exceptions but return values to show whether an exception has occurred. Therefore, when developing the smart contract code, the developer needs to verify the return value of these interfaces to determine whether the call is successful. At the Ethereum virtual machine level, these interfaces send messages to other contract account addresses but cannot throw out the exception, and these interfaces are named Low-Level calls.
Example: As shown in Fig. 9, this function do not check the return value of *callee.call()*.

```
function callnotchecked(address callee) public {
    callee.call();
}
```

Fig. 9. Unchecked call return value.

2.3. Gated graph neural networks

We use gated graph neural network (Li et al., 2016) (GGNN) to learn node-level features. The GGNN is a recurrent graph neural network that updates nodes' states by exchanging neighborhood information recurrently until it reaches the maximum exchange steps. For relieving long-distance dependence issues, GGNN uses a gated recurrent unit to remember the key features. This allows GGNN to go deeper than other GNNs. So this deeper model has a powerful ability to learn more semantics features from graph data.

At the tth state exchanging step in GGNN, each node v will update its state from $h_v^{(t-1)}$ to $h_v^{(t)}$ by aggregating all its neighbors through a gated recurrent unit as

$$h_v^{(t)} = GRU(h_v^{(t-1)}, Agg(\{h_u^{(t-1)} \mid u \in \mathcal{N}(v)\}))$$

where the set $\mathcal{N}(v)$ is neighbors of v and $Agg(.)$ is the aggregation operation, $GRU(.)$ is the gated recurrent unit. By repeating this message exchange for T times, the model can get the final representation for each node as $h_v^{(T)}$.

3. Motivation examples

In this section, we present the motivation behind our smart contract vulnerability detection framework.

Observation 1 (*Joint Code Graph Representation Helps Enhance Vulnerability Detection for Smart Contracts*)**.** Different types of Vulnerabilities in smart contracts are related to distinct features. However, a single type of code graph representation's ability is too limited to reflect different vulnerabilities' complex features comprehensively. So we propose to combine multiple types of graphs to reflect different features in a smart contract function.

Fig. 10 is a function example with a dangerous delegatecall vulnerability, the *_wendy.delegatecall* at line 2 call external contract at address *_wendy*. However, the address *_wendy* is an input parameter that hackers can control. This vulnerability is caused by an insecure data dependency relation between *delegatecall* and the input parameter. So for the function with this type of vulnerability, the key feature that needs to represent is the data dependency relations within each statement.

Fig. 11 is a function with block info dependency vulnerability. The statements from line 10 to line 12 are controlled by *block.number* at line 9. As malicious miners can manipulate *block.number*, this insecure control dependency relations between *block.number* and statements from lines 10 to 12 is the key feature of this vulnerability. It needs a control dependency graph to highlight this feature when vulnerabilities detection.

These two examples show that different vulnerabilities have different characteristics. Moreover, we present some common vulnerabilities and their related features in Table 1. The diversity of vulnerability characteristics leads to the need to combine different code graphs for the code representation. This approach can more comprehensively reflect the code features required for vulnerability detection.

Table 1
Common smart contract vulnerabilities.

Vulnerability name	Description	Related feature
Block Inof dependency	Use blockchain information as a trigger to some critical operation (Luu et al., 2016a; Jiang et al., 2018).	Control dependency
Dangerous delegatecall	The destination of delegatecall is untrust (Parity Technologies, 2017; Jiang et al., 2018).	Data dependency
Unchecked return value	Unchecked the return value of low level call (e.g. address.call) (Luu et al., 2016a).	Data & control dependency
Integer error	Incorrect arithmetic operation (swcregistry, 2019a).	Syntax
Reentrancy	A malicious contract can re-enter the victim contract (swcregistry, 2019b).	Control flow
Short address attack	Accept address with a length less than 20 bits (Zhang et al., 2020).	Data dependency
Suicide contract	Insufficient access control to selfdestruct operation (Nikolic et al., 2018).	Control dependency

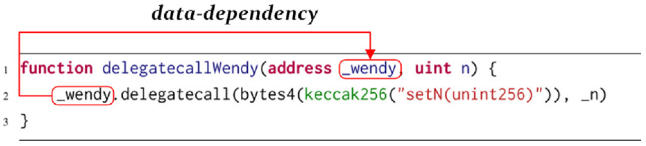

Fig. 10. A function with dangerous delegatecall, the target of delegatecall is an input parameter.

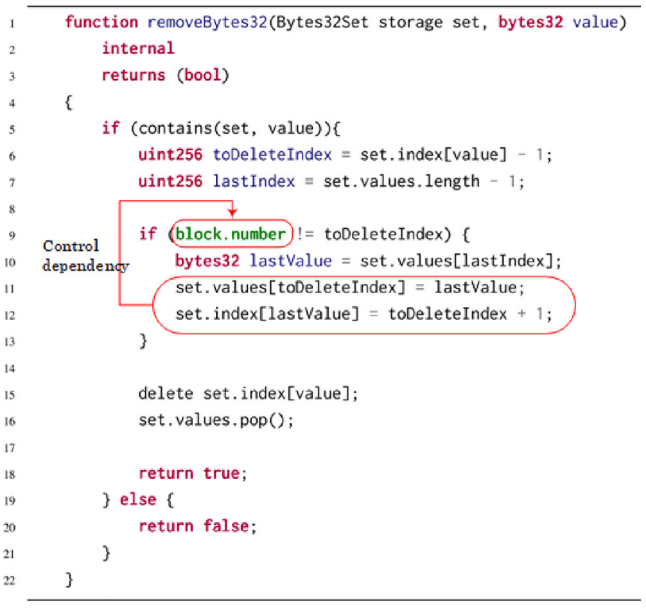

Fig. 11. A block info dependency vulnerability caused by insecure control flow. The statements from line 10 to 12 are controlled by the block.number which can be controlled by miners.

Observation 2 (*Performing Program Slicing to Eliminate Redundant Information Is Necessary*)**.** In a function, only a small part is vulnerable, and others are irrelevant to vulnerability. So these irrelevant statements will be noise to the vulnerability detection task. As shown in Fig. 11, the vulnerability is caused by using *block.number* as the condition to perform state update at lines 10 to 12. Moreover, the control flow of statements from lines 15 to 22 are not related to the *block.number*, so these statements will be noise to detect this vulnerability. Using the whole function as an input to the detection model will extract features from these noise parts to interfere with vulnerability features learning.

As mentioned in Section 2.2, most vulnerabilities in smart contract are related to sensitive operate (e.g. external function call) or sensitive data (e.g. block.number). These sensitive statements are

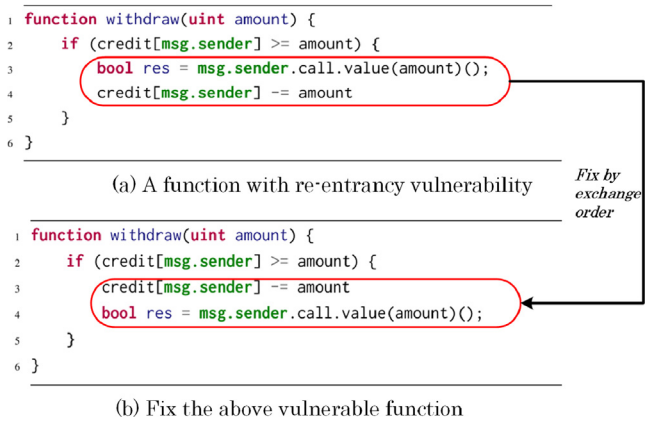

Fig. 12. A re-entrancy vulnerability is caused by the order of the control flow.

only a part of a smart contract function, and others may be noise to vulnerability detection. So we propose it is essential to pure these noise statements as far as possible to enhance the detection result.

Observation 3 (*Contextual Information Is Necessary for Vulnerability Detection*)**.** The contextual information of a statement is essential in the vulnerability detection task. The function in Fig. 12(a) has the reentrancy vulnerability. It first checks whether the credit of the caller stored in this contract *credit[msg.sender]* is enough at line 2. Then transfer the credit from the contract to the caller at line 3. Finally, it deducts the caller's credit record in the contract at line 4. This dangerous deduct-after-transfer control flow order let hacker can re-enter this function and do more transfer than expected. Switching the transfer and deduct order can fix this vulnerability, as shown in Fig. 12(b). An ideal model should have the ability to learn the contextual features of the transfer operation at line 2. So we think that it is necessary for a vulnerability detection model can learn sufficient context information for each statement.

However, the graph neural network generates each node's feature by aggregating its neighbor nodes' features along the direction of edges. As the direction of each edge in code graph representation (e.g. AST, CFG) is unidirectional and follows the code execution order, it makes graph neural network generate each node's feature only by aggregating its precursor nodes' and missing the successor nodes. Therefore, it is necessary to make the model can aggregate features from both precursor and successor nodes.

For all three observations, we utilize Fig. 13 to illustrate how they work in vulnerability detection. Combining the syntax feature of line 3 (*.call.value()*) in AST with the control-and-data

```
1  function confirm(address _h, uint n, byte data) returns (bool)
   ↪ {
2      if (m_txs[_h] != 0) {
3          _h.call.value(n)(data);
4          m_txs[_h] -= n;
5          return true;
6      }else{
7          m_fail += 1
8          return false
9      }
10 }
```

Fig. 13. Smart contract function example.

dependence between line 4 and lines 2–3 in PDG will reveal that this function may contain reentrancy vulnerability (Observation 1). Performing program slicing can eliminate the statements on lines 6–9 irrelevant to reentrancy vulnerability to relieve noise interference (Observation 2). Moreover, the context of line 3 will confirm that it contains the reentrancy vulnerability (Observation 3).

From Observation 1, we propose combining AST, CFG, and PDG as a joint code graph representation to represent a smart contract function that comprehensively captures syntax and semantic features from the source code. From Observation 2, we propose to perform program slicing to reduce the interference from vulnerability irrelevant statements. From Observation 3, we propose doing bidirectional feature learning during Graph Neural Network training, making the model aggregate contextual features for each node.

4. Our approach

We propose an approach to detecting smart contracts vulnerabilities at the slice level. The high-level idea of our approach is to reflect the critical syntax and semantic features from source code by a code graph representation and use the GNN model to inspect any vulnerable patterns inside the code.

Fig. 14 shows the workflow of our approach. Our approach takes the source code files of smart contracts as input. To extract sufficient syntax and semantic information, we first perform graph representation generation on each function of the target smart contract by combining different code graphs (e.g. AST or CFG). Then we perform program slicing to eliminate vulnerability-irrelevant nodes from the generated graph representation. After program slicing, we put the sliced graph into a bidirectional GGNN model for graph feature learning to obtain the hidden feature for each node in the graph. After obtaining each node's feature, we utilize a hybrid attention graph pooling, which combines both self-attention pooling and global attention pooling, to construct the graph-level feature based on each node's feature. Finally, we use a Multilayer Perceptron as the classifier to detect vulnerability from the graph-level feature.

4.1. Joint code graph generation

As mentioned in Observation 1, representing a smart contract function with some code graph alone is not enough to represent a smart contract's vulnerability feature. So we propose a Joint Code Graph Representation (JCG) generated by combining AST, CFG and PDG to reserve sufficient syntax and semantic features from code.

Abstract syntax tree (AST) of a smart contract function is the intermediate representation produced by the compiler. AST reflects syntax features of source code and is the basis for other types of representations. Fig. 15 is the AST of the function in Fig. 13, it is generated by the official solidity compiler solc-js,[1] and it has four types of nodes. The statement node (e.g., ParamList, IfStatement or Expression) represents different statement types. The sub-nodes of IfStatement (e.g., PREDICATE, TRUEBODY or FALSEBODY) can reflect the structure of IfStatement. The operator node (e.g., IndexAccess, != or FunctionCall) represents specific operations in a statement. The leaf nodes in AST are each statement's operands (e.g., m_txs or _h).

Control Flow Graph (CFG) can reflect statements' execution order in a function and conditions of each execution path. In CFG, each node represents a statement and connect by directed edges. As shown in Fig. 17, it is a CFG of the example function in Fig. 13 produced by Slither (Feist et al., 2019). The edge labeled 'Next' reflect the execution order of each statement. Edges labeled by 'False' or 'True' reflect conditions for different execution paths. As mentioned before, the fallback mechanism can let a hacker re-enter the function when a contract starts an external function call operation, so we add a fallback edge from an external function call operation to the function's entry to represent the possible re-entry situation.

Program Dependence Graph (PDG) (Ferrante et al., 1987) can reflect data and control dependency relations among statements in a function. In PDG, there are two types of edges as data dependency edges and control dependency edges. As shown in Fig. 18 is the graph representation for the example with data dependency edges can reflect the data dependency relations of statements and input parameters. Fig. 19 is the graph with control dependency edges which present the control dependency relations of statements and predicates.

For a smart contract function f, we construct a Joint Code Graph Representation(JCG) $G_f = (V, E)$, where $V = \{v_i | v_i \in V_{AST}\}$ that the nodes come from the function's AST. Then add edges from different graphs to this Joint Graph representation that $E = \{e_i | e_i \in E_{AST} \cup E_{CFG} \cup E_{PDG}\}$. As shown in Fig. 16, this JCG by Combining different graphs can enhance the ability of feature representation.

4.2. Program slicing

As discussed in Observation 2, many vulnerability irrelevant nodes may interfere with vulnerability detection. So these redundant nodes have to be eliminated as much as possible. We first define several slice criteria based on the unique syntactic feature from the nine common vulnerabilities in the smart contract. Then program slicing is performed to extract the corresponding vulnerability-related statements by our pre-defined slice criteria. Finally, we pure the JCG by only keeping vulnerability-related nodes and the edges link these nodes.

(1) Slice Criteria: Most smart contract vulnerabilities have unique syntax characteristics such as external function calls or use blockchain information. We define slice criteria based on these vulnerabilities' syntax characteristics to preserve vulnerability-related statements in the source code.

We classify our selected syntax characteristics for slice criteria into two categories: sensitive operations and sensitive data. Sensitive operations such as *.delegatecall* or *.call* will call an external contract may which may be malicious. The wrong usage of sensitive operations in a smart contract is the reason for many vulnerabilities. Sensitive data are the insecure data which hackers can manipulate (e.g. block.number, block.timestamp). These are another main reason for vulnerability in a smart contract. Table 2 is an example of slice criteria for different vulnerabilities.

[1] https://github.com/ethereum/solc-js.

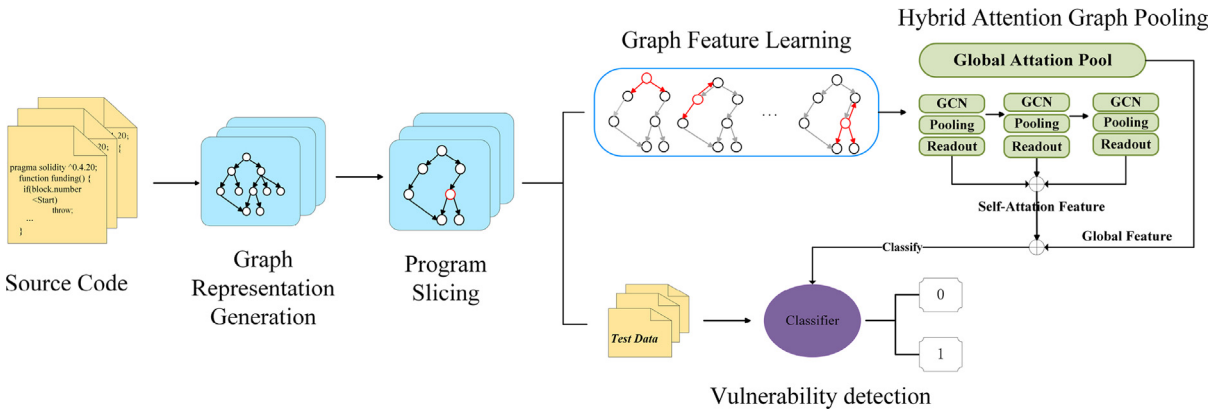

Fig. 14. The workflow of our approach.

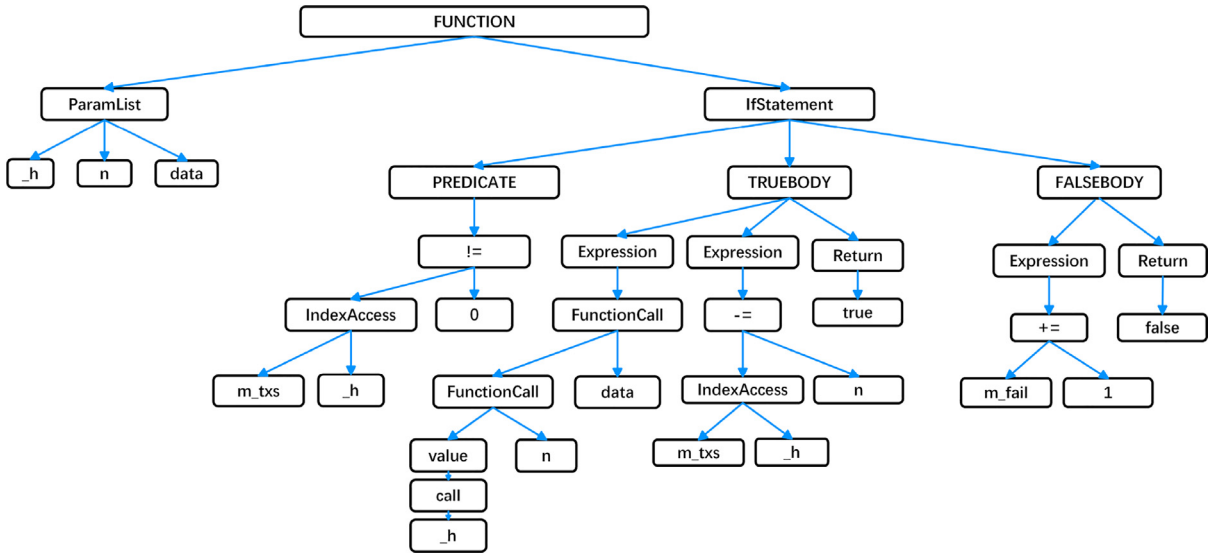

Fig. 15. Abstract syntax tree of Fig. 13.

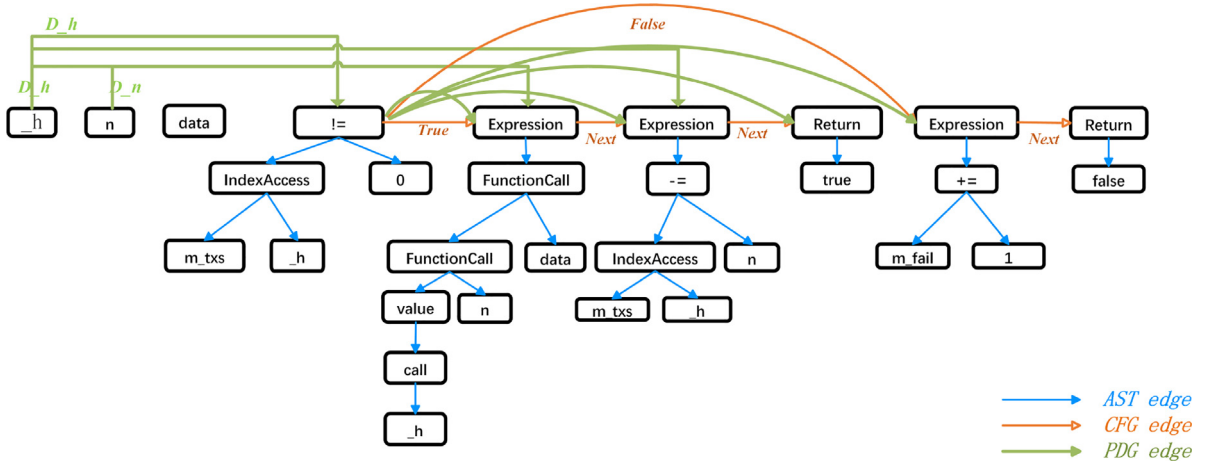

Fig. 16. The joint graph representation of example in Fig. 13.

(2) Sliced-JCG Generation: Based on JCG, we perform program slicing start from nodes that contain slice criteria and do dependency relations analysis to select a set of vulnerability related statements. During dependency analysis, we conduct forward and backward traversal along control and data dependency edges, starting with slice criteria nodes. Furthermore, all the reachable nodes will add to the sliced statements set.

After getting the sliced statements set, all statements in it and their sub-nodes will be reserved in JCG, and other statements

Table 2
The slice criteria for different vulnerabilities.

Vulnerability name	Slice criteria	Slice criteria examples
Block Info dependency	block info	*block.number, block.gaslimit*
Dangerous delegatecall	delegatecall	*address.delegatecall*
Integer overflow	arithmetic operations	$*, +, -$
Re-entrancy	external function call	*address.call*
Suicide contract	selfdestruct operation	*selfdestruct*
Short address attack	input address parameters	*address input_address*
Timestamp dependency	block timestamp	*block.timestamp,now*
Transaction order dependancy	input address parameters	*address input_address*
Unchecked call return value	low level call operation	*address.call*

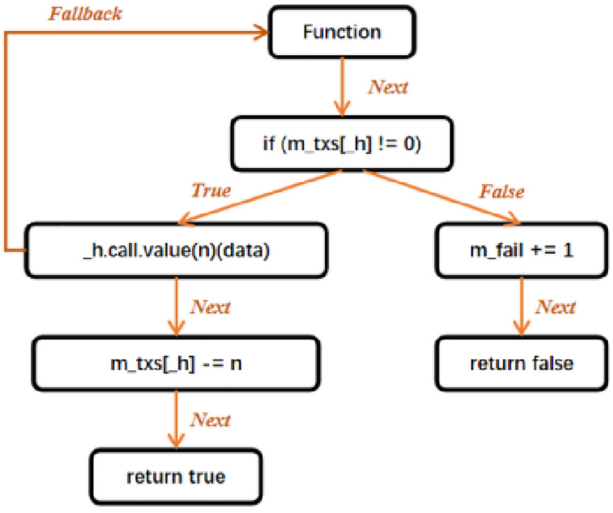

Fig. 17. Control flow graph with fallback edge of Fig. 13.

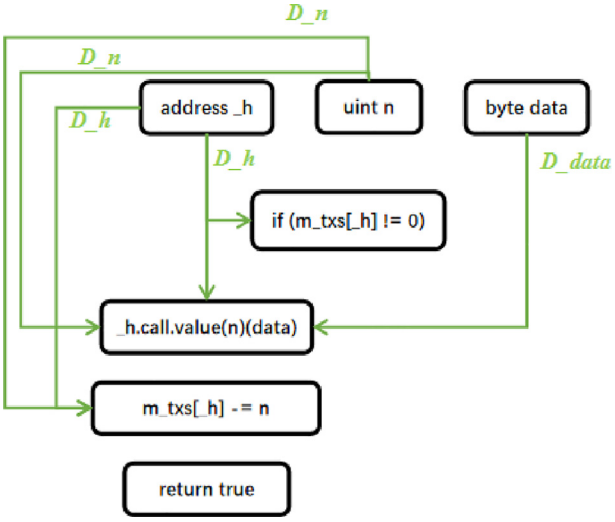

Fig. 18. Data dependency graph of Fig. 13.

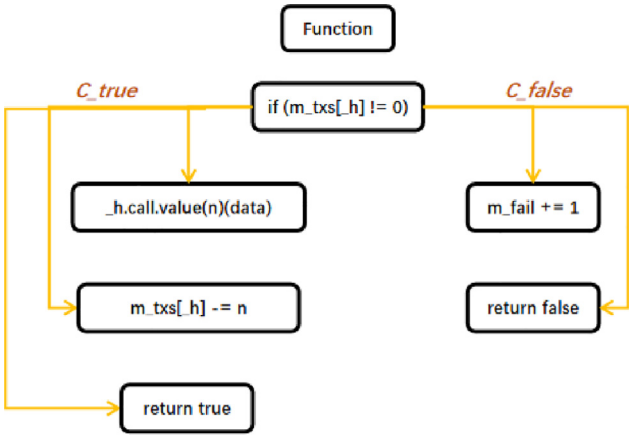

Fig. 19. Control dependency graph of Fig. 13.

```
1  function Collect(uint _am) public payable
2  {
3      var acc = Acc[msg.sender];
4      if( acc.balance>=MinSum && acc.balance>=_am &&
   ↪  now>acc.unlockTime)
5      {
6          if(msg.sender.call.value(_am)())
7          {
8              acc.balance-=_am;
9              LogFile.AddMessage(msg.sender,_am,"Collect");
10         }
11     }
12     else{
13         acc.error = 1
14         LogFile.AddMessage(msg.sender,_am,"Fail");
15     }
16 }
```

Fig. 20. A function with reentrancy vulnerability.

outside the set with all related sub-nodes and edges are removed from JCG to produce a sliced-JCG.

The function in Fig. 20 has a reentrancy vulnerability. The slice criteria *addr.call* is at line 6. From line 6, we do the forward and backward slice to select a set of control or data related to this slice criteria. As shown in Fig. 21, Forward slice select statements at line 3 which is control depended by line 6 and line 4 that data depended by line 3. Backward slice select statements at line 8 and line 9.

4.3. Graph feature learning

After getting the sliced code graph representation of a smart contract function, we transfer it to a vector representation which is the acceptable input form for a deep learning model. We first split each statement into a sequence of tokens. Then we use FastText (Bojanowski et al., 2017) as the embedding model to convert each token into vector representations.

We use the embedding result as the initial node representation for each node, for a node v, the embedding result named $x_v \in \mathbb{R}^d$ and d is the dimension of the embedding vector. Then we use bidirectional GGNN as the model for learning semantic features through the graph structure.

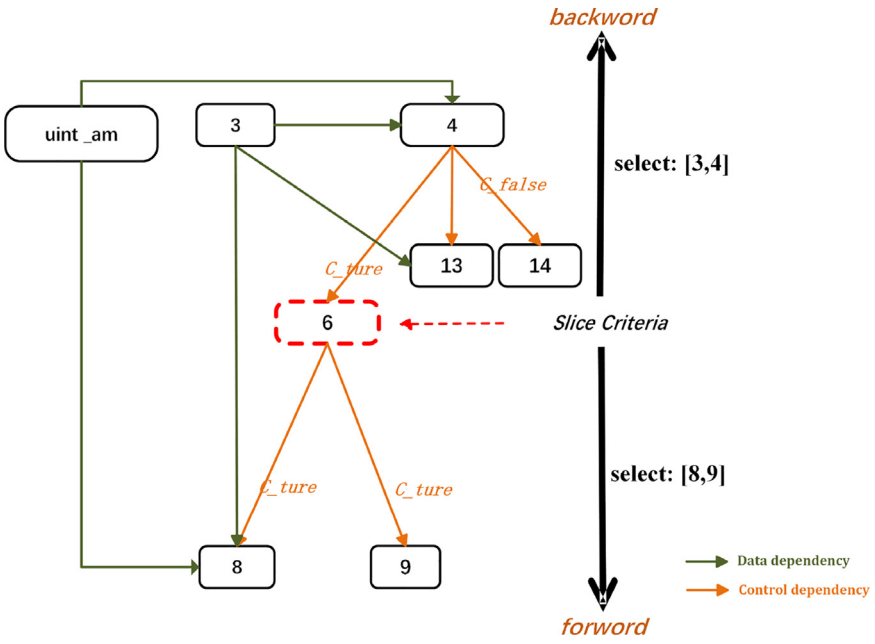

Fig. 21. Program slice result of Fig. 20.

First define the node feature vector for node v at time t is $h_v^{(t)} \in \mathbb{R}^D$, the D is the dimension of the feature vector. As $d < D$, we use the embedding result for each node to initialize feature vector first d dimensions and padding left by 0, so the initial feature vector for each node v is $h_v^{(0)} = [x_v, [0, \ldots, 0]]$.

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

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

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

As mentioned in Observation 3, contextual information is essential for vulnerability detection in the smart contract. However, the GNN update node's feature along the direction of the edge. As shown in (1), the node feature update depends on the adjacency matrix. For a directed graph, the adjacency matrix only contains the outgoing edge for each node. It makes a feature that can only pass forward. So the backward feature is missing during the feature learning process. It weakened the model's ability to learn contextual features of a node.

In this paper, we propose to conduct the adjacency matrix for a graph with both incoming and outgoing edges for each node. For example, a pair of nodes $[v_1, v_2]$ with an direct edge $e_{1,2}$ as an outgoing edge of node $[v_1]$ from v_1 to v_2, but this edge also can be seen as an incoming edge of node v_2. So we combine both outgoing and incoming edges to conduct the adjacency matrix of a graph. It makes the model can learn contextual features for each node from both forward and backward directions.

After getting the aggregated feature $a_v^{(t)}$ for node v at time t, put it into the gated recurrent unit with the node's previous feature vector at time $t-1$ to get its current feature vector.

$$h_v^{(t)} = GRU(a_v^{(t)}, h_v^{(t-1)}) \tag{2}$$

Repeat message aggregating and node feature updating steps for T times, and the model learn each node's final feature $h_v^{(T)}$.

4.4. Hybrid attention graph pooling

After graph feature learning, we get the final feature vector for each node. However, our goal is the vulnerability detection task that predicts the label of a whole graph. Therefore, we need a graph pooling layer to get the entire graph-level feature vector from all nodes' features for the graph-level classification task. Considering that the importance of each node to this graph-level classification task is different, we propose to use the hybrid attention graph pooling that combines self-attention pooling and global attention pooling to generate the final graph-level feature. This hybrid attention mechanism based pooling layer will make the final graph-level feature vector focus more on task-relevant nodes' features.

We first use the hierarchical self-attention graph pooling layer (H-SAGPooling Lee et al., 2019) to generate graph-level features from the most task-related nodes in a graph by self-attention mechanism. As shown in Fig. 22, the self-attention pooling layer is consists of 3 blocks. In each block, it first calculates the attention score by a graph convention layer for each node. Then it picks up the top-k nodes based on the value of the attention score to construct a more task-related subgraph, where the k is a hyper-parameter named pooling rate. Then newly constructed subgraph will be put into an average pooling to generate a subgraph-level feature. Also, this subgraph will be the next block's input to extract a more related subgraph from it. Finally, combining all these three blocks' subgraph-level features, it can get a graph-level feature h_{top-K}.

This H-SAGPooling can extract the most task-related features from all nodes by the attention mechanism, but just selecting top-K nodes from the graph may lose some global information. So we propose to add a global attention pooling layer to complement the whole graph feature by the attention mechanism. In this layer, we propose to conduct the global graph feature by the weighted sum of all nodes features. A soft attention mechanism conducts the weight for each node.

As shown in Eq. (3), we use a Multi-Layer Perceptron (MLP) to compute the attention score for each node's feature and use and a *Softmax* to get the weight for each node. By weighted sum all nodes' features, we get the global graph feature h_{global} for each function.

$$h_{global} = \sum_{i=0}^{N} Softmax(MLP(h_i^{(T)})) * h_i^{(T)} \tag{3}$$

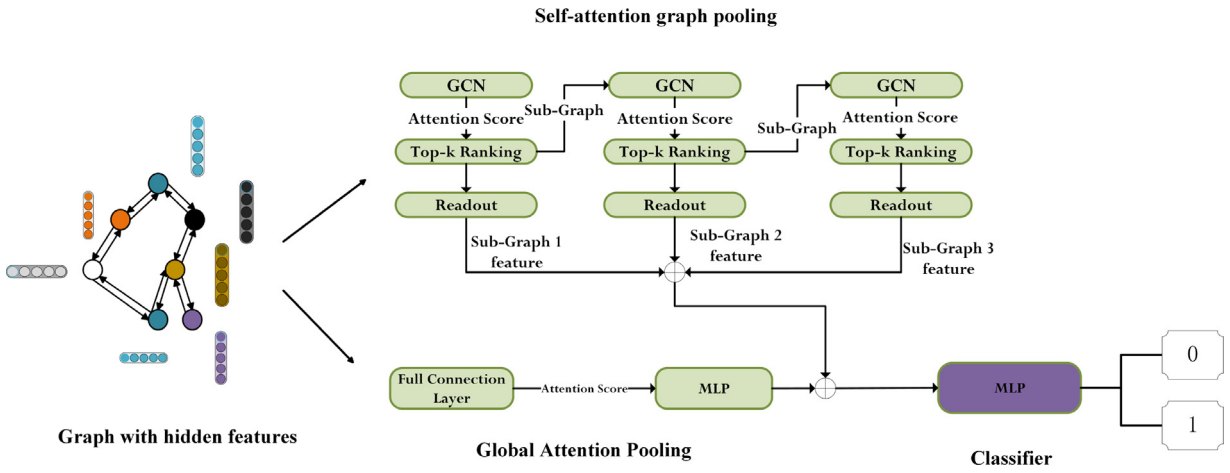

Fig. 22. The hybrid attention pooling layer and classifier for graph classification.

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

4.5. Vulnerability detection

After getting the final graph-level feature vector for a function as h_G, we utilize it as the input with another MLP as a classifier for vulnerability detection, and the detection result can be expressed as $\widetilde{y} = MPL(h_G)$. If the function is vulnerable, the value of $\widetilde{y}$ will be labeled as '1', otherwise labeled as '0'.

5. Experiments

5.1. Research questions

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

RQ1: How effective is our approach in detecting vulnerabilities for smart contracts compared with state-of-the-art vulnerability detection approaches?

The research question is intended to investigate the ability of our approach in smart contract vulnerabilities detection on the same dataset when compared with the state-of-the-art approaches. To answer the question, we compare our approach with a set of state-of-the-art smart contract vulnerability detection work on our dataset, which includes:

- **Oyente** (Luu et al., 2016b): A EVM bytecode level smart contract vulnerability detection tool based on symbolic execution approach. It constructs the CFG (control flow graph) from a smart contract's bytecode then applies symbolic execution on the CFG to check pre-defined vulnerable patterns.
- **Mythril** (Mueller, 2018): A EVM bytecode level smart contract vulnerability detection tool based on concolic analysis, taint analysis, and control flow checking to detect smart contract vulnerabilities.
- **Smartcheck** (Tikhomirov et al., 2018): A source-code-level vulnerability detection tool based on static analysis. It converts AST to a XML parse tree as an intermediate representation, then check pre-defined vulnerability patterns on this intermediate representation.
- **Securify** (Tsankov et al., 2018a): A EVM bytecode level smart contract vulnerability detection tool based on formal verification. It statically analyzes EVM bytecode to infer contract by Souffle Datalog solver to prove some pre-defined safety properties are satisfied.
- **Slither** (Feist et al., 2019): A source-code-level vulnerability detection tool based on static analysis and intermediate representation. It converts a contract to a specific intermediate representation named SlithIR, then doing the pre-defined vulnerability patterns match on this SlithIR representation.
- **Peculiar** (Wu et al., 2021): A deep learning-based source-code-level vulnerability detection tool based on crucial data flow graph and the GraphCodeBERT (a pre-trained model for programming language proposed by Guo et al. (2020)). It converts a contract to a crucial data flow graph and puts it into the GraphCodeBERT model for vulnerability detection.
- **TMP** (Zhuang et al., 2020): A deep learning-based source-code-level vulnerability detection tool based on graph representation and graph neural networks. It converts a contract to a normalized graph and then utilizes a deep learning model named temporal message propagation network (TMP) for vulnerability detection.

RQ2: Can our proposed code graph representation improve the result of vulnerability detection?

Our proposed Sliced-JCG is a code graph representation that combines different code graph representations with bidirectional edges and performs program slicing. To investigate the efficiency of this code graph representation approach for smart contract function vulnerability detection, we conduct ablation experiments on it.

First, we investigate if the program slice can improve the vulnerability detection result by comparing the function-level and slice-level vulnerability detection results. Second, we investigate is the combinations of different cod representations are beneficial to the vulnerability detection task. Finally, we investigate whether our proposed bidirectional graph representation approach improves the vulnerability detection result.

RQ3: Can our proposed GGNN with hybrid attention pooling achieve higher precision results than other pooling layers?

To demonstrate the effectiveness of our hybrid attention pooling that combines hierarchical self-attention and global attention in the vulnerability detection task. We use the Sliced-JCG as the input data and compare the result of GGNN with different types of pooling layers. We use the pooling without attention, such as sum pooling and average pooling as the baseline, then conduct ablation experiments targeting the hybrid attention mechanism. We use the hybrid attention mechanism as a comparison, removing self-attention and global attention, respectively.

5.2. Dataset

Unlike traditional programs with a sufficient vulnerability database like CVE, the community of smart contracts is immature and lacks a mature labeled dataset. So we have constructed a dataset with smart contract solidity source code files based on several existing datasets (Ghaleb and Pattabiraman, 2020; Ferreira et al., 2020).

The first dataset is SolidiFI Benchmark (Ghaleb and Pattabiraman, 2020). This work constructed the dataset by injecting different vulnerabilities into 50 original vulnerability-free contracts to create vulnerable smart contracts. After injection, there are 9369 distinct vulnerable functions in this dataset. The second dataset we used is the HuanGai ManualCheckDataset.[2] It is also a manually constructed benchmark that creates vulnerable smart contracts by vulnerability injection. It supported different vulnerabilities injection, namely Block Informations Dependency, Unchecked Call ReturnValue, Transaction Order Dependency and Unhandled Exception. The third dataset we selected comes from SmartBugs Curated (Ferreira et al., 2020). It is a manually labeled dataset with 143 annotated solidity smart contracts with 208 vulnerable functions. The fourth dataset[3] is also a manually labeled dataset with four types of vulnerabilities. The last dataset we use is SmartBugs Wild (Ferreira et al., 2020), a recently-released, Solidity smart contract dataset crawled from Ethereum. We randomly select 7000 contracts containing the slice criteria were declared in Table 2 and manually labeled 6779 functions which have no vulnerabilities from these contracts.

5.3. Experimental settings

Experiment setup. All the experiments are conducted on a computer equipped with an Intel(R) Core(TM) i5-7300HQ CPU @ 2.50 GHz, a GPU at 1060, and 16 GB of Memory. Our smart contract slice-level vulnerability detection approach comprises graph representation generation, code slice, and model train three parts. In the graph representation generation step, the AST generate tool is implemented with typescript based on solc-typed-ast package.[4] The CFG and PDG is conducted by Slither (Feist et al., 2019). The smart contract slice tool is implemented with python based on the networkx.[5] In the model training step, we use the pre-trained embedding model provided by SmartEmbed (Gao et al., 2020) to do the token embedding, and the GGNN model is implemented with python based on the dgl library (Wang et al., 2019).

Evaluation metrics. We apply the following four widely used evaluation metrics to measure the effectiveness of our approach and the other competitors. ***Accuracy (A)***. The proportion of all samples that are detected correctly. ***Precision (P)***. The proportion of correctly detected vulnerable samples to those detected to be vulnerable. ***Recall (R)***. The proportion of correctly detected vulnerable samples to all vulnerable samples. ***F1-Score (F1)***, A score that measures the overall effect by considering precision and recall.

The trained hyper-parameters settings. The number of time steps is 8; the embedding size for each token is 150; the size of the hidden layer is 200; batch size is 1024; optimizer is ADAMAX (Kingma and Ba, 2014); learning rate is 0.0001; dropout is 0.2; the pool rate for self attention pooling layer is 0.5.

[2] https://github.com/xf97/HuangGai/tree/master/manualCheckDataset.

[3] https://github.com/PaperCodeBase/Dataset.

[4] https://github.com/ConsenSys/solc-typed-ast.

[5] https://github.com/networkx/networkx.

Table 3
Vulnerability detection capability of the different approach.

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Oyente	59.4	39.2	46.1	41.5
Mythril	51.1	62.1	37.5	46.7
Securify	52.9	56.1	53.7	54.8
Smartcheck	41.1	57.7	41.7	48.1
Slither	58.4	62.5	59.5	60.9
Peculiar	80.8	51.9	43.5	47.3
TMP	85.8	82.4	63.8	71.9
Bi-GGNN	**90.2**	**89.2**	**92.9**	**91.1**

6. Experimental results

6.1. Experiments for answering RQ1

Table 3 shows the results of each vulnerability detection in terms of evaluating the metric as mentioned above. As shown in Table 3, Our approach substantially outperforms the existing approach on vulnerability detection.

We notice that these EVM bytecode level detection tools (i.e., Oyente, Mythril, Securify) cannot achieve satisfactory results. The reason is that when compiling a smart contract source code into EVM bytecode, most semantic and syntax features are lost. Such as the control flow graph is difficult to reconstruct from the EVM bytecode due to the *Jump* instruction deciding the control flow of a smart contract and the destination address of *Jump* instruction is not an opcode parameter but dynamically computed by previous code. Therefore, these tools need to symbolically execute the EVM bytecode to compute the jump destination at runtime to construct the control flow using a theorem prover. This makes these detection tools hard to cover all execution paths at the EVM bytecode level. So the missing syntax and semantic make it hard to detect vulnerabilities at the EVM bytecode level.

Compared to source code level detection tools (i.e., slither, smartcheck) based on pre-defined pattern matching, our proposed approach outperforms these. The reason is that these source code level tools are all based on pre-defined patterns to detect vulnerabilities, but these patterns are too simple or fixed to cover all situations. For example, the timestamp dependency vulnerability pattern defined in Smartcheck (Tikhomirov et al., 2018) is looking for expression that contains "==" or "!=" and "block.timestamp" or "now", but this pattern is too simple to detect the actual vulnerability. As shown in Fig. 23, the function calculates a variable by *block.timestamp* at line 2 and 3. Then use the variable as an index to access an array at line 5. As the miner can manipulate *block.timestamp*, the value of *curTimes* can be controlled by a malicious miner. However, there is no expression in Fig. 23 matches the pattern defined in Smartcheck. Our GGNN based model can automatically learn semantic features from the representation of Solidity source code to detect different types of vulnerabilities.

Compared to the learning-based detection approaches (Peculiar Wu et al., 2021 and TMP Zhuang et al., 2020), ours still outperforms these. The reason is that although both Peculiar and TMP utilize the graph to represent the smart contract code like ours. However, neither of their graph representation can comprehensively and precisely capture the vulnerability-related syntax and semantic features inside the code, which may limit their ability in vulnerability detection. For Peculiar (Wu et al., 2021), it only uses the data flow alone to represent the smart contract code, which may lose several critical features such as control flow or syntax information inside the code. Furthermore, not all data in the contract are related to the vulnerability. The flow information of the vulnerability irrelevant data may bing some noise and interfere with the model training. Thus, this

```
function mint(address _to, uint256 _amount) public returns
  ↪ (bool) {
    uint256 curTime = block.timestamp;
    uint256 curTimes = curTime / (31536000);

    if(maxAmountPer[curTimes] == 0) {
        maxAmountPer[curTimes] = totalSupply * (maxProportion)
  ↪ / (100);
    }
    return true;
}
```

Fig. 23. Timestamp dependency vulnerability Missed by SmartCheck.

Table 4
The effect of combining different types of graph representation in vulnerability detection at slice level.

Graph representation	A (%)	P (%)	R (%)	F1 (%)
AST	84.1	83.6	85.8	84.7
AST + CFG	85.8	85.4	90.3	87.9
AST + PDG	88.1	87.0	91.1	89.1
AST + CFG + PDG	**90.2**	**89.2**	**92.9**	**91.1**

one-sided code representation approach makes the deep-learning model hard to learn all potential vulnerability patterns from such insufficient vulnerability-related semantic and syntax features. For TMP (Zhuang et al., 2020), although it somewhat considers the syntax and semantic features inside a smart contract during graph representation. However, it still suffers from the feature missing. In TMP, it only considers control-and-data flow relations during the graph construction and misses the dependency relations. Moreover, TMP simplifies the graph without dependency analysis-based program slicing, making the proposed graph too oversimplified and losing some critical syntax features inside the smart contract code. For example, TMP will simplify the whole *if(**PREDICATE**){**TRUEBODY**}else{**FALSEBODY**}* code block as only one node in the graph with a simple symbol "GB" as the node's content. The detailed implementation inside the **PREDICATE**, **TRUEBODY** and **FALSEBODY**, which contain syntax information, will be lost. Thus, this oversimplified code representation approach in TMP will limit the deep learning model to cover all vulnerability patterns and cause a low recall rate. For our approach, it represents the code of the smart contract by combining AST, CFG and PDG, which can comprehensively capture both syntax and semantic features with few missing and make our approach outperform these two approaches.

Based on the above analysis, our approach outperforms both pattern-based and learning-based comparison approaches. For existing pattern-based approaches, their main limitation is that the manually defined detecting rules cannot cover various vulnerabilities patterns. However, our approach extracts syntax and semantic patterns for different vulnerabilities by utilizing deep learning, a general manner without craft analysis and can adapt to various vulnerabilities patterns. For existing learning-based approaches, the main limitation is their poor code representation ability, which only can reveal partial syntax or semantic features inside the code with several vulnerability-irrelevant components. However, our approach preserves the comprehensive code features (i.e., syntax information or control-and-data dependency relation) in code representation by combining various code graphs (e.g. AST, CFG and PDG) with program slicing, which can comprehensively and precisely cover various common characteristics of different vulnerabilities and improve the detecting performance.

Table 5
The effect of code slicing to vulnerability detection.

Graph representation	A (%)	P (%)	R (%)	F1 (%)
UnSliced JCG	87.6	86.5	88.9	87.7
Sliced JCG	**90.2**	**89.2**	**92.9**	**91.1**

Table 6
The effect of bidirectional graph representation to vulnerability detection.

Graph representation	A (%)	P (%)	R (%)	F1 (%)
Unidirectional graph	88.3	87.8	92.0	89.8
Bidirectional graph	**90.2**	**89.2**	**92.9**	**91.1**

6.2. Experiments for answering RQ2

Table 4 compares the result of slice level vulnerability detection with different code graph representation combinations. The result shows that the combination of all three types of graph representation performs the best among others. Table 5 shows the result of slice level and function level vulnerability detection, in which we observe that slice level vulnerability detection is more effective than function level for smart contracts. Finally, Table 6 shows that the bidirectional code graph representation approach is more fitting for vulnerability detection.

As shown in Table 4, the combination of AST and CFG can get 2% improvement on accuracy, 2.1% on precision, 4.9% and 3.7% on recall and F1-measure than use AST as the code representation alone. It points that the code graph representation combining syntax and semantic features can improve the vulnerability detection result. We also notice that combining AST and PDG is more effective than combining AST and CFG. The reason may be that the PDG can reflect both some part of semantic information of control flow and the data dependence semantic features from the source code, which makes the PDG more effective in code representation. The other reason is that, as mentioned in Observation 1, most vulnerabilities (e.g. Block Info Dependency, Dangerous Delegatecall) are caused by sensitive interface misuse(syntax feature) or dependency on dangerous variables (data or control dependency features), so the PDG is more effective to highlight these vulnerability patterns. Combining different graph representations can reflect different features in smart contract code, and the model will significantly improve. This result proves that the code representation with sufficient syntax and the semantic feature can improve the ability of the model to learn vulnerability features.

From the quantitative results of Table 5, it shows the slice level vulnerability detection outperforms than the function level that it can achieve 3.1% and 4.5% improvement on precision and recall than unsliced function level detection. The reason may be that we use specific smart contract vulnerabilities' syntax characteristics as the slice criteria that can preserve the most vulnerability-related component in the source code. Moreover, the program slicing is based on control and data dependency relations between each statement. Our code graph representation will remove redundant statements that are neither control nor data dependent on these vulnerabilities' syntax characteristics. Removing these noise nodes from the graph allows the model to learn vulnerability-related features with less interference and improve detection results.

Then in Table 6 shows that bidirectional code graph representation improves the detection result. Bidirectional based graph representation achieves 2.2% improvement on accuracy, 1.6% on precision, 1% on recall, and 1.5% on F1-measure than the common unidirectional code graph representation. This bidirectional graph allows the model to learn contextual features for each statement

Table 7
Comparison between different pooling layers.

Pooling layer	A (%)	P (%)	R (%)	F1 (%)
Sum pooling	77.8	76.2	84.3	80.1
Avg pooling	83.8	79.7	90.2	84.7
Global attention pooling	85.2	82.5	91.8	86.9
Self attention pooling	87.9	87.2	91.3	89.1
Hybrid attention pooling	**90.2**	**89.2**	**92.9**	**91.1**

by accommodating its precursor and successor nodes' features which are necessary for vulnerability detection.

Based on the above analysis, we get the following finding that the bidirectional sliced joint code graph representation is more effective in vulnerability detection result.

6.3. Experiments for answering RQ3

The comparison of our model with different pooling layers are shown in Table 7. We can see that our hybrid attention graph pooling layer get significant improvements in smart contract vulnerability detection task.

The experimental results show that using pooling with attention mechanism can be more effective than other no-attention pooling layers. Concretely, the global-attention pooling layer can achieve 1.7% improvement on accuracy, 3.5% on precision, 1.7% on recall, and 2.6% 90.2-7on F1-measure than average pooling layer which has no attention mechanism. The reason is that when generating the final graph feature representation, the attention mechanism can make the model more focused on the most task-related part in the graph.

The comparison of global attention pooling layer, self-attention pooling layer and hybrid attention graph pooling layer shows that the hybrid attention, which combines global attention and self-attention, is more effective than using them alone. As shown in Table 7, the hybrid attention can achieve a 4.8% improvement on recall and 1.5% improvement on precision on average than the other two. The reason is that these two attention mechanisms are complementary, and combining them can improve the effectiveness. The H-SAG use a three graph convolutional network to extract top-k nodes with the highest weight to construct a sub-graph as the final representation. The approach focuses on the local and most task-related components in a graph but drops others with lower weights. The dropped nodes make the model miss some global feature inside the function. Using global attention can reserve the whole graph's feature and supplement the global feature missing from the former. So combining these two attention can let the pooling layer both get the local and global features from the graph.

Based on the above analysis, we find that the hybrid attention graph pooling layer, which combines global attention and self-attention, is more effective in vulnerability detection.

7. Threats to validity

In this section, we discuss the threats to our study, including external threats and internal threats.

External threats. One of the main external threats to our study is the reliability of the dataset. We collected labeled smart contracts written in Solidity from different sources. The first category of our data set comes from other works' data public on Github. These contracts are labeled manually by other researchers. Although they are experts in this area, such a process is inevitably affected by human subjectivity. The second category is a manually constructed dataset by vulnerabilities injection. The injected code logic is relatively simple which may only involve a few obvious vulnerability patterns. Moreover, it only contains nine types of vulnerabilities in our dataset, and we define the corresponding slice criteria for them. It may limit our approach to only can detect all the nine vulnerability types contained in our dataset and hard detecting the type that does not appear in our dataset. However, extending our approach to support the absent type is convenient by adding new samples to the training dataset with corresponding slice criteria through our provided API.

Internal threats. There are also some internal threats in our implementation. First, Many uncertain issues in the data processing and model training phases, such as sample imbalance and hyper-parameter adjustment, may threaten the validity of our approach. With the size of the dataset increasing, more factors need to control in the experiment. It is easy to cause accidental experimental results. We randomly shuffled the dataset in the same proportion and used a validation set to test the model's generalization performance to alleviate this potential threat. Second, we manually define the slice criteria based on the sensitive operations and sensitive data by summarizing the syntax characteristics of different vulnerabilities. It is quite possible that the selected slice criteria are not perfect and may miss some corner cases. Such as, if some vulnerabilities have no dominated syntax characteristics or can be caused by some new syntax feature introduced by Solidity language updating, it is hard for us to enumerate all possible slicing criteria. It will make our approach miss some vulnerability-related semantics during the program slicing. Moreover, if we utilize some redundant slice criteria which may not be directly related to the semantic of a vulnerability, it may cause noise interference during the training phase. Both these situations of inappropriate slice criteria selection may cause a high false-negative rate for our approach. Thus one way to improve our approach is to identify other slice criteria that can reflect additional vulnerability-related semantics and filter irrelevant code components during the model training.

8. Related work

Static Analysis. Static analysis is a general technology that aims at checking for known vulnerable patterns by analyzing the bytecode or high-level code of a smart contract. Slither (Feist et al., 2019), and SmartCheck (Tikhomirov et al., 2018) are vulnerability pattern checker that performs static analyses on source code for many classes of vulnerabilities. Sereum (Rodler et al., 2019) use taint analysis to monitor data flows from storage variables to detect reentrancy vulnerability. SASC (Zhou et al., 2018) construct the topology diagram of the invocation relationship and use it to detect potential vulnerability.

Symbolic Execution. Symbolic execution is a traditional vulnerability mining approach. It is now the most popular approach used in smart contract vulnerability detection. The symbolic execution engine provides a virtual execution environment for the target contract and abstracts the contract's input into symbolic values. An SMT solver is used to solve path constraints to explore program branches as much as possible. Oyente (Luu et al., 2016b) is the first symbolic execution tool combination with Z3 SMT solve (de Moura and Bjørner, 2008) to detect four types of vulnerabilities: transaction ordering dependency, reentrancy and unchecked exceptions. Manticore (Mossberg et al., 2019), Maian (Nikolic et al., 2018) and Teether (Krupp and Rossow, 2018) are symbolic execution tools which support detect composite vulnerabilities in smart contract which are triggered by a sequence of transactions. Osiris (Joon-Wie Tann et al., 2018) combined symbolic execution and taint analysis to detect integer-related vulnerabilities in the smart contract.

Fuzzing. Fuzzing is an efficient software analysis technique that generates a large number of test cases for the target program.

ContractFuzzer (Jiang et al., 2018) randomly generate test cases and use a modified Ethereum client to record execution trace to detect vulnerabilities. ILF (He et al., 2019) is a fuzzing tool enhanced by deep learning to improve the quality of the generated test cases.

Formal verification. Formal verification is an effective way to prove a program meet some security priority. Datalog-based formal verification work such as Vandal (Brent et al., 2018), Securify (Tsankov et al., 2018b), and Ethainter (Brent et al., 2020) use datalog as a domain-specific language to encode security properties and check either these properties are compliant or violated. ZEUS (Kalra et al., 2018) first convert the smart contract to LLVM and use SeaHorn to do the formal verification.

Deep Learning Based. some works use deep learning in smart contract vulnerability detection. SmartEmbed (Feist et al., 2019) determines the presence of vulnerabilities by calculating the similarity to smart contracts with known bugs based on a deep learning model. SMARTEMBED (Gao et al., 2020) use FastText (Bojanowski et al., 2017) model to learn a contract embedding and calculate the similarity to detect vulnerabilities. Joon-Wie Tann et al. (2018) use MAIAN (Nikolic et al., 2018) to label contract and leverage LSTM to predict potential flaws at smart contract's opcode level. Huang (2018) convert contracts' bytecode to RGB colors and then use the CNN model to detect vulnerabilities on a manually labeled dataset. Zhuang et al. (2020) use graph neural networks to defect three types of vulnerabilities in smart contracts, which are reentrancy vulnerability, timestamp dependence vulnerability, and infinite loop vulnerability.

9. Conclusion

This paper proposed a graph neural network-based approach to detect vulnerabilities in smart contracts at the slice-level automatically. Compared to existing approaches, the constructed sliced joint graph representation for smart contract's function reserves sufficient syntactic and semantic information in the source code by combining different types of graph representation and pure the noise nodes by program slicing. We use the bidirectional graph representation for contextual feature learning during the model training phase and a hybrid attention pooling layer to extract graph-level features for vulnerability detection.

In future work, we will improve our study in three aspects. First, we only focus on the combination of three types of graphs. Future research should be conducted by considering more graph representations to increase the performance of vulnerability detection. Second, we evaluate our approach on smart contracts written in Solidity. In the future, we will try to develop a general graph mining approach for different programming languages for smart contracts. Third, we perform vulnerability detection within a function. We will extend our approach to support cross-function vulnerability detection.

CRediT authorship contribution statement

Jie Cai: Conceptualization, Methodology, Software. **Bin Li:** Supervision. **Jiale Zhang:** Data curation, Software, Investigation. **Xiaobing Sun:** Writing – original draft, Validation. **Bing Chen:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62206238, No. 61972335, No. 61872312); the Natural Science Foundation of Jiangsu Province, China (No. BK20220562); the Six Talent Peaks Project in Jiangsu Province, China (No. RJFW-053), the Jiangsu "333" Project, the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University, China (No. KFKT2020B15, No. KFKT2020B16), the Future Network Scientific Research Fund Project, China (FNSRFP-2021-YB-47), the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project, China (YZ2021157, YZ2021158), the Key Laboratory of Safety-Critical Software Ministry of Industry and Information Technology, China (No. NJ2020022), the Natural Science Research Project of Universities in Jiangsu Province, China (No. 22KJB520010, No. 20KJB520024), and Yangzhou University Top-level Talents Support Program (2019), China.

References

Alt, Leonardo, Reitwiessner, Christian, 2018. SMT-based verification of solidity smart contracts.

Bojanowski, Piotr, Grave, Edouard, Joulin, Armand, Mikolov, Tomas, 2017. Enriching word vectors with subword information. Trans. Assoc. Comput. Linguist. 5, 135–146.

Brent, Lexi, Grech, Neville, Lagouvardos, Sifis, Scholz, Bernhard, Smaragdakis, Yannis, 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Donaldson, Alastair F., Torlak, Emina (Eds.), Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. ACM, pp. 454–469.

Brent, Lexi, Jurisevic, Anton, Kong, Michael, Liu, Eric, Gauthier, François, Gramoli, Vincent, Holz, Ralph, Scholz, Bernhard, 2018. Vandal: A scalable security analysis framework for smart contracts. CoRR, abs/1809.03981.

Falkon, S., 2017. The story of the DAO—Its history and consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee.

Feist, Josselin, Grieco, Gustavo, Groce, Alex, 2019. Slither: a static analysis framework for smart contracts.

Ferrante, Jeanne, Ottenstein, Karl J., Warren, Joe D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.

Ferreira, João F, Cruz, Pedro, Durieux, Thomas, Abreu, Rui, 2020. SmartBugs: a framework to analyze solidity smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 1349–1352.

Gao, Zhipeng, Jiang, Lingxiao, Xia, Xin, Lo, David, Grundy, John, 2020. Checking smart contracts with structural code embedding. IEEE Trans. Softw. Eng..

Ghaleb, Asem, Pattabiraman, Karthik, 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 415–427.

Guo, Daya, Ren, Shuo, Lu, Shuai, Feng, Zhangyin, Tang, Duyu, Liu, Shujie, Zhou, Long, Duan, Nan, Svyatkovskiy, Alexey, Fu, Shengyu, et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Hang, Lei, Kim, Do-Hyeun, 2020. Reliable task management based on a smart contract for runtime verification of sensing and actuating tasks in IoT environments. Sensors 20 (4), 1207.

He, Jingxuan, Balunovic, Mislav, Ambroladze, Nodar, Tsankov, Petar, Vechev, Martin T., 2019. Learning to fuzz from symbolic execution with application to smart contracts. In: Cavallaro, Lorenzo, Kinder, Johannes, Wang, XiaoFeng, Katz, Jonathan (Eds.), Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. ACM, pp. 531–548.

Huang, TonTon Hsien-De, 2018. Hunting the ethereum smart contract: Color-inspired inspection of potential attacks. arXiv preprint arXiv:1807.01868.

Jiang, Bo, Liu, Ye, Chan, W.K., 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 259–269.

Joon-Wie Tann, Wesley, Jie Han, Xing, Gupta, Sourav Sen, Ong, Yew-Soon, 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. pp. arXiv–1811, ArXiv E-Prints.

Kalra, Sukrit, Goel, Seep, Dhawan, Mohan, Sharma, Subodh, 2018. ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society.

Kingma, Diederik P., Ba, Jimmy, 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Krupp, Johannes, Rossow, Christian, 2018. TeEther: Gnawing at ethereum to automatically exploit smart contracts. In: Enck, William, Felt, Adrienne Porter (Eds.), 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. USENIX Association, pp. 1317–1333.

Lee, Junhyun, Lee, Inyeop, Kang, Jaewoo, 2019. Self-attention graph pooling. In: International Conference on Machine Learning. PMLR, pp. 3734–3743.

Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, Zemel, Richard S., 2016. Gated graph sequence neural networks. In: Bengio, Yoshua, LeCun, Yann (Eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings.

Luu, Loi, Chu, Duc-Hiep, Olickel, Hrishi, Saxena, Prateek, Hobor, Aquinas, 2016a. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.

Luu, Loi, Chu, Duc-Hiep, Olickel, Hrishi, Saxena, Prateek, Hobor, Aquinas, 2016b. Making smart contracts smarter. In: Weippl, Edgar R., Katzenbeisser, Stefan, Kruegel, Christopher, Myers, Andrew C., Halevi, Shai (Eds.), Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. ACM, pp. 254–269.

Mossberg, Mark, Manzano, Felipe, Hennenfent, Eric, Groce, Alex, Grieco, Gustavo, Feist, Josselin, Brunson, Trent, Dinaburg, Artem, 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE, pp. 1186–1189.

de Moura, Leonardo Mendonça, Bjørner, Nikolaj, 2008. Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, Jakob (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. In: Lecture Notes in Computer Science, vol. 4963, Springer, pp. 337–340.

Mueller, Bernhard, 2018. Smashing ethereum smart contracts for fun and real profit. p. 54.

Nakamoto, Satoshi, 2008. Bitcoin: A peer-to-peer electronic cash system. Decentralized Bus. Rev. 21260.

Nikolic, Ivica, Kolluri, Aashish, Sergey, Ilya, Saxena, Prateek, Hobor, Aquinas, 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. ACM, pp. 653–663.

Osborne, Charlie, 2021. DAO maker crowdfunding platform loses $7M in latest DeFi exploit. https://www.zdnet.com/article/cream-finance-wallet-pilfered-for-34-million-in-cryptocurrency/.

Parity Technologies, 2017. A postmortem on the parity multi-sig library self-destruct. https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct.

Park, Ji-Sun, Youn, Taek-Young, Kim, Hye-Bin, Rhee, Kyung-Hyune, Shin, Sang-Uk, 2018. Smart contract-based review system for an IoT data marketplace. Sensors 18 (10), 3577.

Pierro, Giuseppe Antonio, Tonelli, Roberto, Marchesi, Michele, 2020. An organized repository of ethereum smart contracts' source codes and metrics. Future Internet 12 (11), 197.

Qian, Peng, Liu, Zhenguang, He, Qinming, Zimmermann, Roger, Wang, Xun, 2020. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access 8, 19685–19695.

Rodler, Michael, Li, Wenting, Karame, Ghassan O., Davi, Lucas, 2019. Sereum: Protecting existing smart contracts against Re-entrancy attacks. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society.

swcregistry, 2019a. Integer Overflow and Underflow, https://swcregistry.io/docs/SWC-101.

swcregistry, 2019b. Reentrancy, https://swcregistry.io/docs/SWC-107.

Szabo, Nick, 1997. Formalizing and securing relationships on public networks. First Monday.

Tann, A., Han, Xing Jie, Gupta, Sourav Sen, Ong, Yew-Soon, 2018. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. pp. 1371–1385, arXiv preprint arXiv:1811.06632.

Tikhomirov, Sergei, Voskresenskaya, Ekaterina, Ivanitskiy, Ivan, Takhaviev, Ramil, Marchenko, Evgeny, Alexandrov, Yaroslav, 2018. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. pp. 9–16.

Tsankov, Petar, Dan, Andrei, Drachsler-Cohen, Dana, Gervais, Arthur, Buenzli, Florian, Vechev, Martin, 2018a. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.

Tsankov, Petar, Dan, Andrei Marian, Drachsler-Cohen, Dana, Gervais, Arthur, Bünzli, Florian, Vechev, Martin T., 2018b. Securify: Practical security analysis of smart contracts. In: Lie, David, Mannan, Mohammad, Backes, Michael, Wang, XiaoFeng (Eds.), Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, on, Canada, October 15-19, 2018. ACM, pp. 67–82.

Wang, Minjie, Zheng, Da, Ye, Zihao, Gan, Quan, Li, Mufei, Song, Xiang, Zhou, Jinjing, Ma, Chao, Yu, Lingfan, Gai, Yu, Xiao, Tianjun, He, Tong, Karypis, George, Li, Jinyang, Zhang, Zheng, 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315.

Wood, Gavin, Reitwiessner, C, Beregszaszi, A, Hirai, Y, et al., 2014a. The solidity contract-oriented programming language.

Wood, Gavin, et al., 2014b. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap. 151 (2014), 1–32.

Wright, Turner, 2021. DAO maker crowdfunding platform loses $7m in latest DeFi exploit. https://cointelegraph.com/news/dao-maker-crowdfunding-platform-loses-7m-in-latest-defi-exploit.

Wu, Hongjun, Zhang, Zhuo, Wang, Shangwen, Lei, Yan, Lin, Bo, Qin, Yihao, Zhang, Haoyu, Mao, Xiaoguang, 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 378–389.

Zhang, Yuanyu, Kasahara, Shoji, Shen, Yulong, Jiang, Xiaohong, Wan, Jianxiong, 2018. Smart contract-based access control for the internet of things. IEEE Internet Things J. 6 (2), 1594–1605.

Zhang, Pengcheng, Xiao, Feng, Luo, Xiapu, 2020. A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 139–150.

Zhou, Ence, Hua, Song, Pi, Bingfeng, Sun, Jun, Nomura, Yashihide, Yamashita, Kazuhiro, Kurihara, Hidetoshi, 2018. Security assurance for smart contract. In: 9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018. IEEE, pp. 1–5. http://dx.doi.org/10.1109/NTMS.2018.8328743, URL https://doi.org/10.1109/NTMS.2018.8328743.

Zhuang, Yuan, Liu, Zhenguang, Qian, Peng, Liu, Qi, Wang, Xiang, He, Qinming, 2020. Smart contract vulnerability detection using graph neural network. In: Bessiere, Christian (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020. ijcai.org, pp. 3283–3290.

Jie Cai received the B.E. degree from Nanjing University of post and telecommunications, Nanjing, China, in 2016. He is currently working toward the Ph.D. degree in software engineering with Yangzhou University, Yangzhou, China. His research interests include blockchain security, software safety and security, etc.

Bin Li received the B.S. degree in computer software from Fudan University, Shanghai, China, in 1986, and the M. S. and Ph.D. degrees in computer application technology from Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1993 and 2001 respectively. He is currently a Professor with Yangzhou University, Yangzhou, China. He has published more than 100 journal articles and conference papers. His main research interests include knowledge graph, software data mining, and multi-agent systems.

Jiale Zhang received the Ph.D. degree in computer science and technology the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2021. He is currently a Lecture with the School of Information Engineering, Yangzhou University, Yangzhou, China. His research interests are mainly federated learning, blockchian security, and privacy-preserving.

Xiaobing Sun received the B.S. degree in computer science and technology from Jiangsu University of Science and Technology, Zhenjiang, China, in 2007, and the Ph.D. degree from the School of Computer Science and Engineering, Southeast University, Nanjing, China, in 2012. He is currently a Professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. His research interests include software maintenance and evolution, software repository mining, and intelligence analysis. He has been authorized more than 20 patents. He has published more than 80 papers in refereed international journals.

Bing Chen received the B.S. and M.S. degrees in computer engineering from the Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing, China, in 1992 and 1995, and the Ph.D. degree in computer technology from the College of Compute Science and Technology, NUAA, in 2008. He is currently a Full Professor with the College of Compute Science and Technology NUAA. His research interests include cloud/edge computing, security and privacy, federated learning, and wireless communications.