DeepFusion: Smart Contract Vulnerability Detection Via Deep Learning and Data Fusion

Hanting Chu, Pengcheng Zhang, *Member, IEEE*, Hai Dong, *Senior Member, IEEE*, Yan Xiao, and Shunhui Ji

***Abstract*—Given that smart contracts execute transactions worth hundreds of millions of dollars daily, the issue of smart contract security has attracted considerable attention over the past few years. Traditional methods for detecting vulnerabilities heavily rely on manually developed rules and features, leading to the problems of low accuracy, high false positives, and poor scalability. Although deep learning-inspired approaches were designed to alleviate the problem, most of them rely on monothetic features, which may result in information incompetence during the learning process. Furthermore, the lack of available labeled vulnerability datasets is also a major limitation. To address these issues, we collect and construct a dataset of five labeled smart contract vulnerabilities, and propose *DeepFusion*, a vulnerability detection method that fuses code representation information, including program slice information and abstraction syntax tree (AST) structured information. First, we develop automated tools to extract contract vulnerability slicing information from source code, and extract structured information from source code-converted AST. Second, code features and global structured features are fused into the data. Finally, the fused data are input into the Bidirectional Long Short-Term Memory+ Attention (BiLSTM+ATT) model for smart contract vulnerability detection. The BiLSTM model can capture long-term dependencies in both directions and is more suitable for processing serialized information generated by *DeepFusion*, while the attention mechanism can highlight the characteristic information of vulnerabilities. We conducted experiments via collecting a real smart contract dataset. The experimental results show that our method significantly outperforms the existing methods in detecting the vulnerabilities of *reentrancy*, *timestamp dependence*, *integer overflow and underflow*, *Use tx.origin for authentication*, and *Unprotected Self-destruct Instruction* by 6.36%, 6.42%, 16.5%, 21.29%, and 25.05%, respectively. To the best of our knowledge, the latter two vulnerabilities are the first to be detected using deep learning methods.**

***Index Terms*—Abstraction syntax tree (AST), data fusion, program slicing, smart contract, vulnerability detection.**

Received 20 March 2023; revised 3 August 2024; accepted 7 October 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62272145 and Grant U21B2016, in part by CloudTech-RMIT Green Bitcoin Joint Research Program, in part by the Fundamental Research Funds for the Central Universities at Sun Yat-sen University under Grant 24qnpy153, and in part by the National Natural Science Foundation of China under Grant 62402499. Associate Editor: W. Chu. *(Corresponding author: Pengcheng Zhang.)*

Hanting Chu, Pengcheng Zhang, and Shunhui Ji are with the College of Computer Science and Software Engineering, Hohai University, Nanjing 211100, China (e-mail: pchzhang@hhu.edu.cn).

Hai Dong is with the School of Computing Technologies, RMIT University, Melbourne, VIC 3000, Australia (e-mail: hai.dong@rmit.edu.au).

Yan Xiao is with the School of Computing, NUS University, Kent Ridge, Singapore 119077 (e-mail: dcsxan@nus.edu.sg).

Digital Object Identifier 10.1109/TR.2024.3480010

I. Introduction

BLOCKCHAIN is essentially a distributed shared transaction ledger, the concept of which was first proposed by Nakamoto in 2008 [1]. Blockchain technology is decentralized, tamper-proof, and traceable. Smart contracts, among the most successful applications of blockchain technology, have garnered significant attention from both academia and industry [2], [3], [4]. A smart contract is a computer program that runs on a blockchain platform, usually written in Solidity, a Turing-complete high-level programming language [5]. Anyone can write a smart contract and publish it to run on Ethereum. The security of a smart contract relies on the safety of its supporting platform Ethereum and its Solidity code. In the past, security issues with smart contracts have caused significant financial losses. In the DAO [6] incident that occurred in June 2016, attackers exploited a reentrancy vulnerability in the smart contracts to make off with over 3.6 million Ether.

Smart contracts are insecure and there are three main reasons why they can be easily attacked by attackers: 1) *Economic performance*. Smart contracts usually handle and manipulate transactions related to encrypted digital currency. For attackers, attacking smart contracts can bring them huge economic benefits. 2) *Programming language*. The application scenarios of smart contract are complex and dynamic. At present, the programming language of smart contract is still novel and rough [7]. In the real-world scenario, it may be difficult for contract developers to conduct tests, resulting in asset security problems of smart contracts. 3) *Deployment platform*. Different from traditional languages, smart contracts cannot be modified once deployed. The existence of security vulnerabilities in smart contracts will lead to unexpected behavior in contracts, which goes against the original intention of creating fair and reliable contracts. Vulnerability detection is a crucial topic in smart contract security research [8]. Many tools have been developed for detecting vulnerabilities [9], [10], [11], [12], [13], [14], [15], [16]. At present, most of the research work on vulnerability detection rely on traditional methods, that is, manually defining and summarizing vulnerability rules according to the characteristics of vulnerabilities to be detected [17], [18]. However, such manually developed detection rules may lead to high false positive rates and are unable to cope with complex vulnerability patterns [19]. Furthermore, with the explosive growth of smart contracts, the detection rules developed by domain experts are less flexible and adaptable to diversity and dynamic changes of smart contract vulnerabilities [20].

Compared with these methods, smart contract vulnerability detection based on deep learning has higher accuracy and completeness. It mainly employs deep learning to learn and analyze the lexical, syntax, control flow, data flow, and other information of the code [18]. However, these deep learning-based vulnerability detection methods are still in the early stage. Most of the existing methods, such as *TMP* and *DR-GCN* [21], rely on single graph features to process the vulnerability programs, which is deficient in a global structured view (e.g., the structure of AST (Abstract Syntax Tree)). This limitation prevents these methods from further improving their detection performance. To solve the above problems, we propose *DeepFusion*, a method combining structured information from AST with program slices. Specifically, *DeepFusion* first analyzes the data flow, control flow of a smart contract, and adopts the program slicing technique to automatically extract vulnerability code slices. Next, the syntax parser is used to compile the smart contract to generate AST. *DeepFusion* then employs the custom traversal method to automatically extract the AST structured information of the contract. Finally, based on a predefined deep learning model, *DeepFusion* can fuse the characteristics of contract vulnerability slice information and AST structured information. It is able to detect five serious contract vulnerabilities in the Ethereum smart contract environment.

Contributions: The main contributions of this article are as follows:

1) We combine sliced information with AST structured information to construct the data fragments that highlight vulnerabilities while preserving the overall data structure and enhancing the interpretability of the data.
2) *DeepFusion* is the first method to detect *tx.origin* vulnerability and *unprotected self-destruct instruction* vulnerability. To this end, we propose a simple and effective feature fusion-based detection network model based on BiLSTM+ATT. To the best of our knowledge, this is the first time such a technique is employed in this area.
3) The experiments show that *DeepFusion* outperforms the other existing methods in terms of vulnerability detection *accuracy*, *recall*, *precision*, and *F1 score*. In the case of *Reentrancy*, *Timestamp Dependency*, *Use tx.origin*, *Integer Overflow and Underflow* and *Unprotected Self-Destruct Instruction* vulnerabilities, the highest *accuracy* of the existing methods is increased by 6.36%, 6.42%, 21.29%, 16.5%, and 25.05%, respectively. *DeepFusion* achieves an average accuracy, recall, precision, and F1 score of 90.74%, 90.80%, 90.29%, and 90.65% for the five vulnerabilities. Our collected dataset and implementations of *DeepFusion* are released at https://github.com/Tourneso/DeepFusion.

The rest of this article is organized as follows: Section II provides background information pertinent to this research, encompassing smart contract vulnerability detection, vulnerability classification frameworks, types of vulnerabilities, and call graphs. Section III details our approach. Section IV demonstrates the validation of our method using a collected dataset of buggy smart contracts. After a discussion of related work in Section V, the article concludes with future work plans in Section VI.

II. Background

A. Vulnerability Classification Framework

Numerous studies have addressed the classification and standardization of smart contract vulnerabilities [22]. In 2017, Atzei et al. [23] were pioneers in analyzing Ethereum smart contract vulnerabilities, classifying them into three categories: programming language, virtual machine, and blockchain [24]. Subsequently, Dika et al. [25] adopted this classification, categorizing security issues into low, medium, and high risks. The decentralized application security project (DASP) in 2018 identified ten types of high-risk vulnerabilities [26]. Two years later, Chen et al. [27] analyzed real Ethereum smart contracts and discussion posts, defining 20 types of flaws concerning security, usability, maintainability, and reusability. In addition, Zhang et al. [28] extended the *IEEE Standard Classification for Software Anomalies* to develop *JiuZhou*, a comprehensive framework that categorizes and assigns severity levels to 49 types of vulnerabilities.

This article focuses on the detection of five kinds of smart contract vulnerabilities selected upon their high severity and/or prevalence, i.e., *Reentrancy* vulnerability, *Timestamp Dependency* vulnerability, *Use tx.origin for authentication* vulnerability and *Integer Overflow and Underflow* vulnerability and *Unprotected Self-Destruct Instruction* vulnerability.

B. Vulnerability Type

Many factors need to be taken into account in the process of selecting the types of vulnerability to be detected. It needs to consider not only a vulnerability's extent of damage, but also its universality [12]. Therefore, we select the following five vulnerabilities as the research targets.

Reentrancy: The *reentrancy* vulnerability is one of the most destructive types of security vulnerabilities. The infamous DAO attack [29], [30] exploited this vulnerability. It typically occurs when a transfer is made to an external user address, and the external user address recursively calls the same function of the contract [19]. Solidity language includes a default fallback function without a function name or parameters, which is automatically triggered when a transfer occurs in a smart contract. During the transfer operation, a *reentrancy* attack can be triggered before the contract state variables are modified. Consequently, an attacker can trigger a malicious fallback function, causing the smart contract to repeatedly call the transfer function until the account balance reaches 0.

Timestamp Dependency: The Ethereum protocol stipulates that miners can freely set the timestamp of the block when the timestamp difference is less than 900 s [25]. If the smart contract developer uses the timestamp as one of the key execution conditions for executing a transaction or generating random numbers, there are risks of timestamp dependence vulnerabilities. When the smart contract uses the timestamp as the execution condition

of the ether transmission, attackers can form an attack by manipulating the timestamp in the block to disguise the transaction, resulting in the loss of property.

Integer Overflow and Underflow: The integer overflow and underflow vulnerability occurs when the result of an operation exceeds the value range of the function itself [31], which results in an out-of-bounds value. Smart contracts issuing ethers are prone to integer overflow and underflow vulnerabilities. There have been numerous cases where smart contracts are exploited by attackers to launch attacks due to integer overflow and underflow vulnerabilities, which is a high-risk vulnerability [32]. An attacker may use this vulnerability to transfer a large number of ethers to a specific address at the cost of a relatively small amount of ethers, which may result in serious financial loss.

Use tx.origin for authentication: The *tx.origin* variable in the Solidity language is a global variable that represents the address of a contract initiating a transaction [33]. For example, if contract A initiates a transaction that calls contract B, and contract B calls contract C, the entire call chain can be abstracted as A $\rightarrow$ B$\rightarrow$C, then *tx.origin* is the address of contract A and msg.sender is the address of contract B. Use *tx.origin* for authentication refers to the use of the *tx.origin* variable to determine the control authority of a contract instead of using the *msg.sender* variable. An attacker may use this vulnerability to disguise their identity and launch an attack against the user who invokes the contract.

Unprotected Self-Destruct Instruction: Since smart contracts are immutable postdeployment, they often include a destruction function, necessitating the incorporation of a suicide function during the contract's development [34]. If a defect is identified within the contract, financial losses can be mitigated by activating the suicide function, which terminates the contract and reallocates the ethers to a predetermined address. However, if a smart contract with a suicide function lacks adequate permission controls, any user can execute this function, prematurely ending the contract and redirecting its ethers to an address of their choosing. Such vulnerabilities can compromise the contract's intended functionality and lead to significant financial losses.

C. Call Graph

A call graph (also known as call multigraph) [35] is a control flow graph that represents the call relationship between subprograms of a computer program. A simple application of call graphs is to find procedures that have never been called. Call graphs can be used for the human to understand a program. They can also serve as a basis for other types of analysis, such as the analysis of value flow between tracking processes, or changing impact prediction. Call graphs can also be used to detect program execution exceptions or code injection attacks. In this article, we analyze the calling relationship between internal functions of a contract based on the call graph generated by *Slither*, so as to obtain the possible data flow and control flow information.

III. DEEPFUSION

A. Overview of DeepFusion

DeepFusion is a deep learning-based vulnerability detection tool for Ethereum smart contracts, which adopts a supervised learning paradigm. It can detect five types of severe vulnerabilities. Fig. 1 shows the overall workflow of *DeepFusion*. *DeepFusion* includes three parts: data preprocessing and representation, model training, and model prediction. *DeepFusion* first conducts experiments using the manually collected, annotated, and verified dataset collected by *Data Collection* described in Section III-B. Second, for different vulnerability types, *DeepFusion* extracts the corresponding target function fragments (described in Section III-C: *Target Function Extractor*). On the basis of the above function fragments, *DeepFusion* analyzes the data flow and control flow of the vulnerable contracts, and extracts the vulnerability program slicing information (implemented by *Program Slicing Extractor* described in Section III-D) and AST structured information (obtained by *AST Structured Information Extractor* introduced in Section III-E). *DeepFusion* fuses the two kinds of information, and then enters the information into a predesigned model for training and prediction (implemented by *Model Training and Model Prediction* depicted in Section III-F).

B. Data Collection

Before detailing the proposed vulnerability detection approach, we outline the data collection process. The dataset was sourced in two primary ways: initially, the keywords *smart contract vulnerability, smart contracts buggy,* and *smart contracts defects* were employed to identify relevant data on GitHub and the Gitter chat room.[1] Subsequently, *Karl*,[2] a tool used in conjunction with *Mythril* for blockchain monitoring, was utilized to gather new smart contracts from the Ethereum.[3] *Karl* and *Mythril* can pinpoint addresses of smart contracts potentially harboring vulnerabilities. For data labeling, we utilized tools such as *Mythril*, *Oyente*, *Slither*, and *SmartCheck* to evaluate the collected contracts. If at least three out of these four tools indicated an issue, a contract line was manually verified and labeled as vulnerable. These verified and labeled contracts were then incorporated into the vulnerability dataset. This process involved two blockchain security developers, each with over five years of experience in smart contract development and security auditing. They reviewed the code to confirm the presence of vulnerabilities and their specific types, thereby ensuring the accuracy of the labels.

C. Target Function Extractor

The goal of target function extractor is to obtain function fragments related to vulnerabilities. *DeepFusion* analyzes the global generated by *Slither* to extract complete function-call paths from a contract. A function-call path refers to a sequence of nodes, each of which denotes a function. The linear relationship in this function-call path represents that the previous function calls the next function. It is noteworthy that for different types of vulnerabilities, the internal workflows and the employed techniques of target function extractor are distinct.

[1] [Online]. Available: https://gitter.im/orgs/ethereum/rooms/
[2] [Online]. Available: https://github.com/cleanunicorn/karl
[3] [Online]. Available: https://etherscan.io

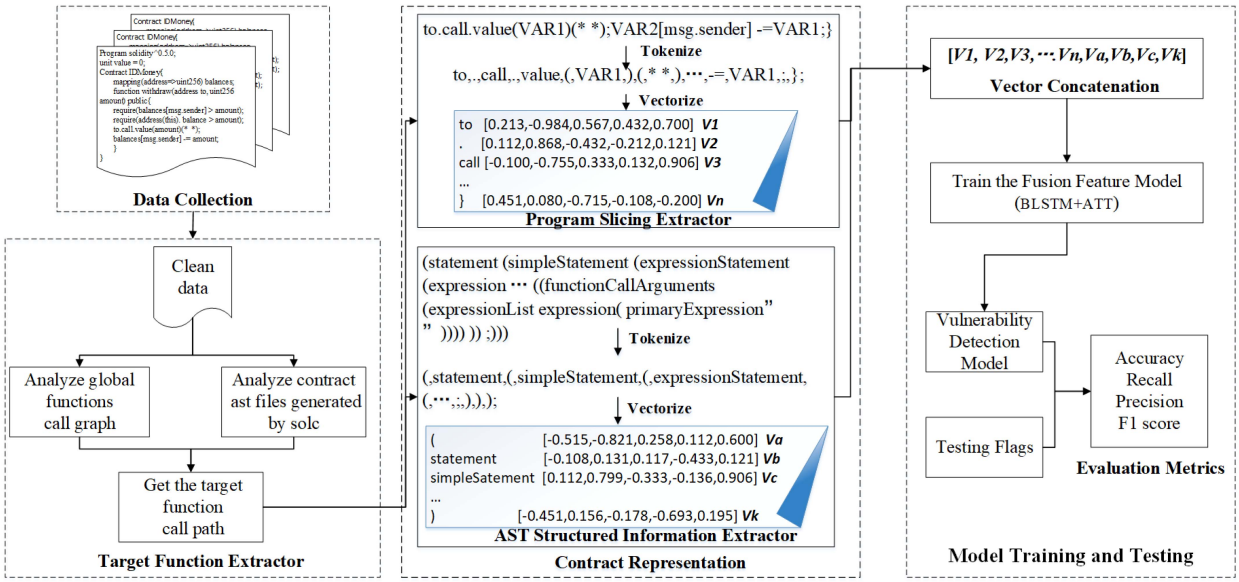

Fig. 1. Overall flowchart for *DeepFusion*.

1) Target Function Extractor of Reentrancy (TFER): *Reentrancy* vulnerability is considered as an invocation to *call-statements* that can call back to itself through a chain of calls. On the basis of the global call-graph, *TFER* analyzes the abstract syntax tree json file generated by *solc* (official compiler of solidity) to locate the function path containing the *call-statement*, *TFER* extracts function paths containing *call-statements*. If there are statements that perform checks on account balance information, *TFER* will extract these statements. For example, if the user balance is deducted before using call.value for transfer, then the contract usually does not suffer from the *reentrancy* vulnerability. In addition, for the objectivity of the dataset, the deposit paths in which a function is declared as *payable* will be extracted for the extraction of the full function fragment containing the transaction. *TFER* subsequently extracts the complete functional fragments in accordance with the specified function paths.

2) Target Function Extractor of Timestamp Dependence (TFET): *Timestamp Dependence* vulnerability exists when a smart contract uses the *block.timestamp* or *now* as part of the attributes to perform critical operations, then the miners can control the attributes related to mining and blocks. If the functions of the contract depend on these attributes, the miner can interfere with the functions of the contract. *TFET* extracts function paths containing *block.timestamp* or *now* in the collected dataset. If there is a variable calling on the value of *block.timestamp* or a statement passing *block.timestamp* as an argument, *TFET* will extract the statement. For example, if *block.timestamp* is assigned to a variable, which is used or restricted by strict condition statements, e.g., *require* or *assert* in the next process, we consider that this contract does not contain timestamp dependence. *TFET* subsequently extracts the complete functional fragments in accordance with the specified function paths.

3) Target Function Extractor of Integer Overflow and Underflow (TFEI): In solidity, an overflow or underflow occurs when the value of an integer variable is higher or lower than it can handle, which can lead to unpredictable situations. Therefore, it is important to validate the input parameters and output results before *arithmetic operations* are performed in a smart contract. *TFEI* first searches for function paths containing the *arithmetic operations-statements* in the contracts. If there is a safemath library function that constrains *arithmetic operations* or a conditional statement that constrains variables of *arithmetic operations*, *TFEI* will extract these constraint statements. For example, if a contract's function fragments containing *arithmetic operations* are bound by the safemath library, we consider that this contract does not contain integer overflow and underflow. *TFEI* subsequently extracts the complete functional fragments in accordance with the specified function paths.

4) Target Function Extractor of Use Tx.origin *(TFETX):* *TFETX* first searches for function paths containing the *tx.origin-statements* in the contracts. If there are *tx.origin-statements* in a contract's function *modifier*), all function fragments modified by this function *modifier* will become our target functions. If *msg.sender==tx.origin* is used to reject an external contract to invoke the current contract, we will identify this function as the target function. *TFETX* subsequently extracts the complete functional fragments in accordance with the specified function paths.

5) Target Function Extractor of Unprotected Self-Destruct Instruction (TFEU): The self-destruct and suicide functions in smart contracts are the methods that can be used to remove an already deployed contract on Ether. The self-destruct and suicide functions can be called to stop the contract when there is a security problem. If the access to these functions is not well protected, then the smart contract can be compromised by an attacker. *TFEU* first search for paths containing a *suicide-statement* or a *selfdestruct-statement*. If there is a variable calling on the value of a *suicide-statement* or a *self-destruct-statement*, *TFEU* also extracts the statement. For example, if a *suicide-statement* or a

Algorithm 1: Program Slicing Extractor Algorithm.

Input: Target functional fragments' source code: SC;
Vulnerability standard: VS
Output: Vulnerability program slices: P
1 ContractAstSet C = getContractASTBySolc(SC) ;
2 VariableSet VC = getSpecificVariable(SC) according to VS ;
3 **foreach** *contractAst in* C **do**
4 **if** *contractAst has data dependence with* VC **then**
5 statementSet dd = getSourceCodeStatement(contractAst) ;
6 p append dd
7 **end**
8 **if** *contractAst has control dependence with* VC **then**
9 statementSet cd = getSourceCodeStatement(contractAst);
10 p append cd
11 **end**
12 **end**
13 return P;

self-destruct-statement is assigned to a variable, which is used or restricted by strict condition statements, e.g., *require* or *assert* in the next process, we consider that this contract does not contain unprotected self-destruct instruction. *TFEU* subsequently extracts the complete functional fragments in accordance with the specified function paths.

D. Program Slicing Extractor

Techniques of program slicing [36] are utilized to extract the semantic information from the code. Program Slicing Extractor is designed to extract program slicing information related to vulnerabilities. Here, we mainly use the abstract syntax tree json file generated by *solc* to extract the vulnerability program slicing information. Algorithm 1 shows the whole process of Program Slicing Extractor.

Given target functional fragments' source code SC and associated vulnerability standards VS, we intend to obtain a program slice of a contract for the vulnerability. First, we use *solc* to compile the contract to generate AST and then generate the variable set VC of the contract based on the vulnerability criteria in *Line 1-2*. In Line 3, we traverse the contract AST. For each statement that has data dependence on the VC, all of them are added to the program slice set as the extracted program slice information of the contract as shown in Lines 4–7. Moreover, for each statement that has control dependence on the VC, all of them are also added into the program slice in Lines 8–11.

1) Program Slicing Extractor of Reentrancy (PSER): For the *reentrancy* vulnerability, *PSER* initially extracts the *address* variable located ahead of the *call-statements* from the target functional fragments described in Section III-C-1. It then forms program slices with all statements that operate on the *address* variable.

Fig. 2 shows a Reentrancy program before and after slicing. First, PSER locates the *call.value* function on line 165 and adds the *transfer* (157, 172) to the set of slicing. It then extracts all statements that contain *_to* variable (158, 159, 161, 164, 165, 166, 167, 170) preceding the *call.value* function. In particular, if the variable is present in a conditional judgment statement (e.g., in an *if*), PSER extracts the entire conditional judgment (161, 168, 169, 171). Finally, the entire set of statements is combined as (157, 158, 159, 161, 164, 165, 166, 167, 168, 169, 170, 171, 172).

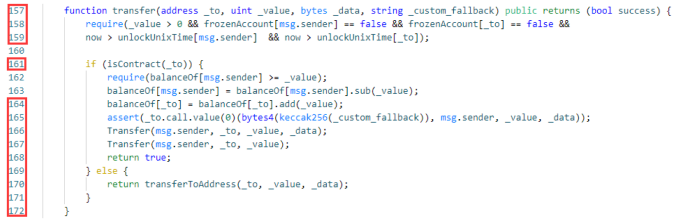

Fig. 2. Comparison before and after program slicing of a reentrancy vulnerability example (slicing information is preserved as circled in red).

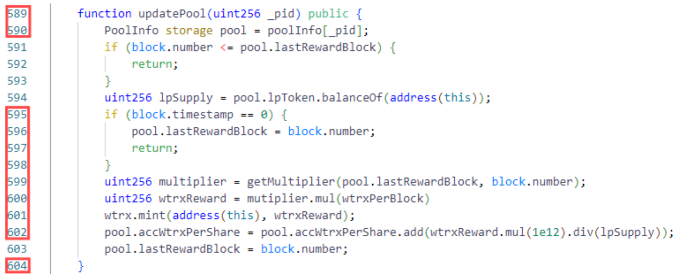

Fig. 3. Comparison before and after program slicing of a timestamp dependence vulnerability example (slicing information is preserved as circled in red).

2) Program Slicing Extractor of Timestamp Dependence (PSET): For the *timestamp dependence* vulnerability, PSET first locates the position of *block.timestamp* or *now*, and then checks whether the variable is assigned to another variable. Next, combined with the AST json file generated by *solc*, PSET analyzes the statements that use the variable in the next function fragment, followed by composing the above statements into slices.

A comparison of a timestamp dependence vulnerability example before and after slicing is shown in Fig. 3. First, PSET locates the variable *block.timestamp* on line 595, and adds the *updatePool* function (589, 604) to this set of slicing. But the variable is located in the *if-statement*. PSET thus adds the whole *if-conditional* judgment code block to this set (595, 596, 597, 598). In the *if-conditional* code block, the next step is to find the statements that have the data dependence with *pool.lastRewardBlock* (599, 600, 601, 602). Finally, the entire set of timestamp dependence is jointed with (589, 590, 595, 596, 597, 598, 599, 600, 601, 602, 604).

3) Program Slicing Extractor of Integer Overflow and Underflow (PSEI): For the *integer overflow and underflow* vulnerability, PSEI first locates the position of *arithmetic operations*, and then checks if any other variable makes a call to that variable. Next, combined with the AST json file generated by *solc*, PSEI analyzes the statements that use the variable in the next function fragment, followed by composing the above statements into slices.

Fig. 4 shows an Integer Overflow and Underflow program before and after slicing. First, PSEI extracts the *arithmetic operations* on line 257 and locates the *batchTransfer* function (255,

```
255  function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
256    uint cnt = _receivers.length;
257    uint256 amount = uint256(cnt) * _value;
258    require(cnt > 0 && cnt <= 20);
259    require(_value > 0 && balances[msg.sender] >= amount);
260
261    balances[msg.sender] = balances[msg.sender].sub(amount);
262    for (uint i = 0; i < cnt; i++) {
263        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
264        Transfer(msg.sender, _receivers[i], _value);
265    }
266    return true;
267  }
```

Fig. 4. Comparison before and after program slicing of an integer overflow and underflow vulnerability example (slicing information is preserved as circled in red).

```
271  function queueTransaction(address target, uint256 value, string memory signature, bytes memory data, uint256 eta) public returns (bytes32){
272      require(eta >= getBlockTimestamp().add(delay), "Timelock::queueTransaction: Estimated execution block must satisfy delay.");
273      require(tx.origin == admin, "Timelock::queueTransaction: Call must come from admin.");
274
275      bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
276      queuedTransactions[txHash] = true;
277
278      emit QueueTransaction(txHash, target, value, signature, data, eta);
279      return txHash;
280  }
```

Fig. 5. Comparison before and after program slicing of a tx.origin vulnerability example (slicing information is preserved as circled in red).

268), then finds all the variables associated with this *arithmetic operations* (258, 259, 262). It extracts the statements that contain these variables and are not bound by the *safeMath* function. If the variables appear in a loop statement, PSEI extracts the entire loop (262, 263, 264, 265). Finally, the whole set of statements is combined as (255, 257, 258, 259, 262, 263, 264, 265, 266, 267).

4) Program Slicing Extractor of Use Tx.origin *(PSETX):* For the *Use Tx.origin* vulnerability, PSETX first locates the position of *tx.origin-statements*. If *tx.origin* is used as a variable of *address* type, PSETX obtains all statements that operate with this *address* variable. Next, combined with the AST json file generated by *solc*, PSETX analyzes the statements that use the variable in the next function fragment, and forms these statements into slices.

A comparison of a tx.origin vulnerability example before and after slices is shown in Fig. 5. First, PSETX locates the variable *tx.origin* on line 273. Since this variable is in the *require-statement* judgment, PSETX adds all subsequent statements (275, 276, 278, 279). Finally, it extracts all statements that contain variables (271, 273, 275, 276, 278, 279, 280) and combines the whole set of statements into a new slice.

5) Program Slicing Extractor of Unprotected Self-Destruct Instruction (PTSU): For the *unprotected self-destruct instruction* vulnerability, PTSU first locates the position of *suicide-statement* or *self-destruct-statement*, and then checks whether the variable is assigned to another variable. Next, combined with the AST json file generated by *solc*, PTSU analyzes the statements that use the variable in the next function fragment, and forms these statements into slices.

A comparison of an unprotected self-destruct instruction vulnerability example before and after slices is shown in Fig. 6. First, PTSU locates the variable *self-destruct* on line 233. Since this variable is in the *if-statement* judgment, PTSU inserts subsequent statements (233, 234). Finally, it extracts all statements that contain variables (228, 229, 230, 231, 233, 234) and combines the whole set of statements into a new slice.

```
219  function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
220    initDaylimit(_daylimit);
221    initMultiowned(_owners, _required);
222  }
223
224    function init() public {
225        initialized = 1;
226    }
227
228    function run(uint256 input) {
229        if (initialized == 0) {
230            return;
231        }
232
233        selfdestruct(msg.sender);
234    }
235  }
```

Fig. 6. Comparison before and after program slicing of an unprotected self-destruct instruction vulnerability example (slicing information is preserved as circled in red).

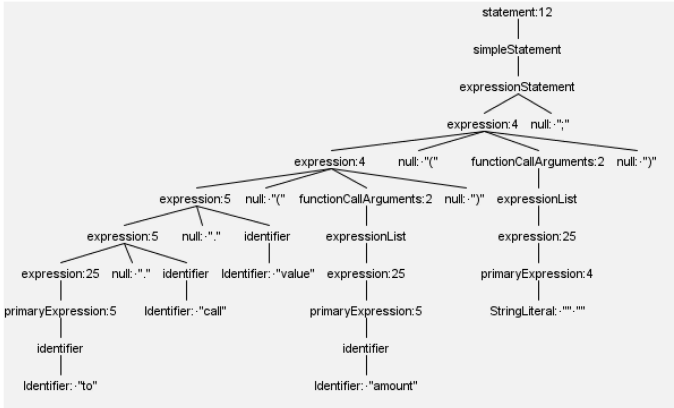

Fig. 7. Example of an abstract syntax tree.

TABLE I
AST INFORMATION SERIALIZATION

(statement (simpleStatement (expressionStatement (expression (expression (expression (expression(expression (primaryExpression (identifier to))) . (identifier call)) . (identifier value))((functionCallArguments (expressionList (expression (primaryExpression (identifier amount))))))) ((functionCallArguments (expressionList (expression (primaryExpression ″ ″)))))) ;)))

E. AST Structured Information Extractor

The AST Structured Information Extractor is designed to extract the structured information from the functional fragments discussed in Section III-C. To achieve this, we employ ANTLR,[4] a powerful parser generator that is adept at translating structured text and analyzing language, to parse these function fragments into an AST [30], [37]. The abstract syntax structure of the source code is represented by the AST in a tree-like format. In this process, the AST of the function fragments is serialized using a depth-first search. It should be noted that, for different types of vulnerability fragments in Section III-C, we use the same traversal method to convert them into serialized AST fragments. Fig. 7 shows a sample abstract syntax tree generated by the ANTLR compilation. The results converted and shown in Table I demonstrate that the structured information is preserved.

[4][Online]. Available: https://github.com/solidity-parser/antlr

```
function transfer(address _to, uint _value, bytes _data, string _custom_fallback) public returns (bool success) {
require(_value > 0 && frozenAccount[msg.sender] == false && frozenAccount[_to] == false &&
now > unlockUnixTime[msg.sender]  && now > unlockUnixTime[_to]);
if (isContract(_to)) {
balanceOf[_to] = balanceOf[_to].add(_value);
assert(_to.call.value(0)(bytes4(keccak256(_custom_fallback)), msg.sender, _value, _data));
Transfer(msg.sender, _to, _value, _data);
Transfer(msg.sender, _to, _value);
return true;
} else {
return transferToAddress(_to, _value, _data);
}
}
```

Fig. 8. Example without normalization.

```
function transfer(address VAR1, uint VAR2, bytes VAR3, string VAR4) public returns (bool VAR5) {
require(VAR2 > 0 && VAR6[msg.sender] == false && VAR6[VAR1] == false &&
VAR7 > VAR8[msg.sender]  && VAR7 > VAR8[VAR1]);
if (FUN1(VAR1)) {
VAR9[VAR1] = VAR9[VAR1].FUN2(VAR2);
assert(VAR1.call.value(0)(bytes4(FUN3(VAR4)), msg.sender, VAR2, VAR3));
Transfer(msg.sender, VAR1, VAR2, VAR3);
Transfer(msg.sender, VAR1, VAR2);
return true;
} else {
return FUN4(VAR1, VAR2, VAR3);
}
}
```

Fig. 9. Example with normalization.

F. Model Training and Model Prediction

1) Normalization and Embedding: Before inputting the data of Section III-D into the model training and tesing, we need to normalize the function fragments to eliminate the impact of some variables. Especially, we focus on the following variables. For user-defined variable names, function names are uniformly represented by VAR$\{n\}$, Fun$\{n\}$, etc., where n represents the order in which variables appear in the current function segments.

As shown in Fig. 8, for user-defined variables, it is valueless for performing vulnerability feature extraction. Instead, we normalize them to reduce the interference of semantically irrelevant information and obtain the corresponding code slice word vector. The data processed by Normalization are shown in Fig. 9.

We need to use word segmentation to convert the program slicing of Section III-D and the AST structured fragments of Section III-E into code tokens, and then use word embedding to convert the above tokens into vectors, which are ready to be inputted into the model for training and testing.

2) Our Model: The work of Dam et al. [38] has verified the advantages of LSTM in code, since code follows a logical and semantic structure and is closely coupled. In addition, vulnerable code fragments is usually closely related to their context, which can be handled by LSTM. Therefore, here we briefly introduce the employed LSTM model and its variant (i.e., BiLSTM). The LSTM model, designed to alleviate the gradient vanishing problem in traditional RNNs, was devised to more accurately find and exploit long-range context using special memory cells. An LSTM layer is composed of several memory blocks, each recurrently connected. Within each block, there are one or more recurrently connected cell states C_t and a single model unit. This unit includes an input gate i_t, an output gate o_t, and a forget gate f_t, all of which regulate the flow of information within the memory block. The forget gate determines which information should be discarded from the cell state, while the input gate decides the amount of new information to be added to the cell state. The output gate determines the value to be generated as output. The hidden state h_t of an LSTM cell is calculated according to (1)–(6). In these equations, $\tilde{C}_t$ represents a vector of new candidate values, σ denotes the sigmoid function, $\tanh$ is the hyperbolic tangent function, and $*$ indicates both matrix multiplication and element-wise product. We employed a bidirectional LSTM (BiLSTM) model to obtain both past and future bidirectional information of sequences, where the sequence features fed into the BiLSTM model are multidimensional. Since the traditional LSTM model does not evaluate the contribution of different sequence features to the final result and weights them equally, we further introduced an attention mechanism to assign different weights to the features based on their contributions

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{1}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{4}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{5}$$

$$h_t = o_t * \tanh(C_t) \tag{6}$$

$$s_t = \tanh(W^\top h_t + b) \tag{7}$$

$$a_t = softmax(s_t) = \frac{\mathrm{Exp}(s_t)}{\sum_t \mathrm{Exp}(s_t)} \tag{8}$$

$$T = \sum_{t=1}^{n} a_t * h_t \tag{9}$$

where (7) is the formula for calculating the score of each feature vector. After capturing the score of each feature vector, (8) obtains the normalized weights for the attention mechanism by performing a normalization operation using the softmax function and T indicates the final output.

3) Training and Prediction: In this article, we integrate two features containing vulnerability information. One is the program slicing feature of vulnerabilities, and the other is the structural feature of serialized AST converted by a vulnerability fragment. Therefore, in the training of our model, the word vectors from the two types of information described in Sections III-D and III-E are concatenated to form a new vector. This vector represents a combined code representation of the feature information. Unlike more complex feature fusion approaches that involve assigning specific weights to each feature set, our method leverages the complementary nature of program slicing and AST features, as program slicing captures control and data flow relationships while AST captures hierarchical structure. Ultimately, this vector is used as the input to train a deep-learning model for detection. We construct a BiLSTM+ATT model. The BiLSTM employs a bi-directional LSTM model to obtain sequence information in both directions, which can better handle long sequences. The addition of an attention mechanism can highlight important features. These models take vectors as input, followed by a standard training process. We conduct experimental analysis in the next section.

For model prediction, given source code of target smart contracts, it will be handled by aforementioned steps (i.e., target

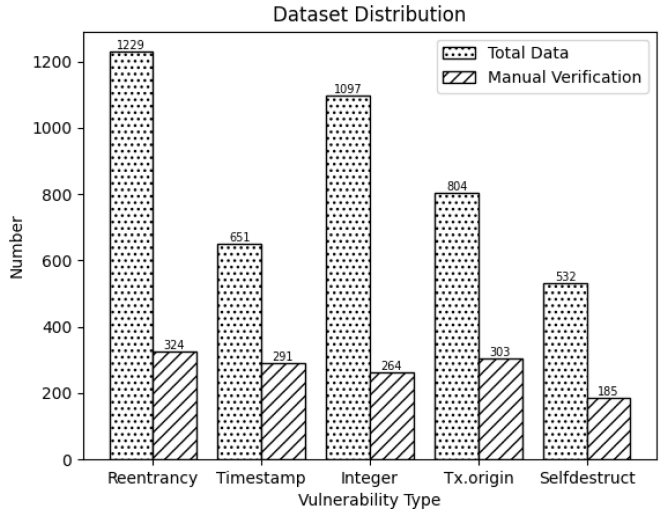

Fig. 10. Dataset distribution.

function extractor, program slicing extractor, AST Structured Information Extractor, and normalization and embedding). The trained models will take these embedded vectors as input to identify whether the target contract contains vulnerabilities.

IV. Experiment and Evaluation

A. Dataset and Research Questions

We manually collected and constructed a dataset containing tens of thousands of smart contracts and performed manual validation. The dataset contains 109 834 smart contracts with about 3 236 421 functions. Through manual analysis and review, 4313 smart contracts containing *Reentrancy*, *Timestamp Dependence*, *Tx.origin*, *Integer Overflow, and Underflow* and *Unprotected Self-Destruct Instruction* vulnerabilities are selected. We used this dataset to evaluate the proposed method. Filtered 1229 smart contracts containing *Reentrancy* functions, of which 324 required manual verification. A total of 651 smart contracts with *Timestamp Dependency* function, of which 291 require manual verification. A total of 804 smart contracts with *Tx.origin* features, including 303 smart contracts that require manual verification. A total of 1097 smart contracts with *Integer Overflow and Underflow* feature, of which 264 smart contracts require manual verification. A total of 532 smart contracts with *Unprotected Self-Destruct Instruction* feature, of which 185 require manual verification. As shown in Fig. 10, the program slicing and model training part were implemented in python, and the AST structured information extraction part was implemented in Java. We randomly divided 80% of the dataset into a training set and the remaining 20% into a test set. We repeated each experiment ten times and calculated the average values to obtain the final experimental results.

We have attempted to respond to the following four research questions:

1) *RQ1:* Is *DeepFusion* applicable to these five vulnerability types? How is its performance?
2) *RQ2:* Can fusion of vulnerability feature fragments and structured information improve the detection performance?
3) *RQ3:* How is the performance of *DeepFusion* compared to the existing methods and detection tools?
4) *RQ4:* How efficient is *DeepFusion*?

B. Evaluation Metrics

We evaluate the performance of *DeepFusion* in term of four widely used indicators, i.e., accuracy, recall, precision, and F1 score, where *TP* indicates that a vulnerable sample is successfully predicted as a vulnerable sample by the model; *FP* indicates that a nonvulnerable sample is predicted as a vulnerable sample by the model; *TN* represents that a nonvulnerable sample is successfully predicted as a nonvulnerable sample by the model; and *FN* represents that a vulnerable sample is predicted as a nonvulnerable sample by the model. The specific calculation formulas are as follows.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$Precision = \frac{TP}{TP + FP} \tag{12}$$

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{13}$$

C. Performance

DeepFusion uses BiLSTM+ATT for model training and detection. To answer *RQ1*, we conduct comparative experiments to detect five types of vulnerabilities by replacing BiLSTM+ATT with LSTM, GRU, DNN, BiLSTM, and RNN to show the effectiveness of the method. First, by comparing among models, the most suitable model for smart contract vulnerability detection is obtained. After selecting the best neural network model, we conduct comparative experiments with different experimental parameters to select the most appropriate hyperparameters. To ensure the fairness of the experiments, each method adopts the same data processing and feature extraction methods, and detects the same vulnerability types.

The range of learning rate is set as [0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01], the range of dropout rate is set as [0.1, 0.2, 0.3, 0.4, 0.5], the range of batch size is set as [32, 64, 128], and the range of the dimension of each token vector is set to [50, 100, 150, 200].

We comprehensively evaluate the performance of the models in terms of accuracy, recall, precision, and F1 score, the results of which are shown in Table II. In terms of processing long sequence data, the performance of the LSTM and DNN model are better than that of the traditional RNN and GRU model. Therefore, LSTM and DNN are more suitable than RNN and GRU for processing smart contract data. BiLSTM can simultaneously capture bidirectional (i.e., forward and backward) long-term and short-term dependencies, which shows better performance than LSTM and DNN. Further, attention mechanism

TABLE II
PERFORMANCE EVALUATION OF THE MODELS ON *REENTRANCY*, *TIMESTAMP DEPENDENCE*, *TX.ORIGIN*, *INTEGER OVERFLOW, AND UNDERFLOW* AND *UNPROTECTED SELF-DESTRUCT INSTRUCTION*

Methods		RNN	GRU	LSTM	DNN	BLSTM	BiLSTM+ATT
Reetrancy	Acc(%)	74.70	83.74	84.33	83.49	86.75	**90.84**
	Recall(%)	81.71	87.96	86.59	87.60	84.94	**89.49**
	Precision(%)	71.28	81.93	82.56	82.35	88.32	**91.84**
	F1(%)	76.14	84.51	84.52	84.90	86.38	**90.65**
Timestamp Dependency	Acc(%)	77.29	81.67	82.50	82.88	87.50	**89.17**
	Recall(%)	65.83	68.75	68.33	72.96	87.92	**89.37**
	Precision(%)	86.63	85.66	85.37	84.91	87.50	**89.20**
	F1(%)	74.28	78.89	79.61	78.49	87.53	**89.28**
Integer Overflow and Underflow	Acc(%)	79.47	80.95	82.64	83.39	86.45	**91.67**
	Recall(%)	75.66	83.87	83.75	86.11	84.36	**88.89**
	Precision(%)	72.15	78.79	76.41	78.73	85.27	**94.14**
	F1(%)	73.86	81.25	79.91	82.26	84.81	**91.43**
Tx.origin	Acc(%)	77.19	81.56	82.50	82.14	89.84	**91.56**
	Recall(%)	82.50	81.04	83.13	86.21	88.13	**93.65**
	Precision(%)	74.95	76.53	79.80	80.19	85.32	**89.74**
	F1(%)	78.43	79.15	82.18	83.09	86.68	**91.65**
Unprotected Selfdestruc Instruction	Acc(%)	75.69	76.19	81.59	83.58	86.57	**90.48**
	Recall(%)	71.83	78.13	79.75	82.21	87.54	**93.75**
	Precision(%)	77.51	75.76	76.02	78.36	85.16	**88.24**
	F1(%)	74.56	76.92	77.84	80.24	86.34	**90.31**

The bold values represent the best performance results.

TABLE III
PERFORMANCE EVALUATION BETWEEN SINGLE FEATURES AND FUSED FEATURES

Methods	Program Slice				AST				Data Fusion			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
Reentrancy	88.36	83.33	87.22	85.23	85.42	81.78	84.78	83.25	**90.84**	**89.49**	**91.84**	**90.65**
Timestamp	85.83	80.67	79.45	80.06	78.33	83.33	75.76	79.36	**89.17**	**89.37**	**89.20**	**89.28**
Tx.origin	86.88	89.00	80.19	84.37	82.19	83.75	84.76	84.25	**91.56**	**93.65**	**89.74**	**91.65**
Integer Overflow	85.47	79.67	92.31	85.53	88.39	87.56	81.59	84.47	**91.67**	**88.89**	**94.12**	**91.43**
Selfdestruct	85.71	93.54	80.56	86.57	80.95	83.87	78.79	81.25	**90.48**	**93.75**	**88.24**	**90.31**

can highlight key features and greatly improve the performance of the BiLSTM model. As a result, BiLSTM+ATT achieves the best performance on most of the metrics.

Answer to RQ1:*DeepFusion* is applicable and effective to the five types of vulnerabilities, and it achieves the best performance on BiLSTM+ATT. *DeepFusion* achieves an average accuracy, recall, precision and F1 score of 90.74%, 91.03%, 90.63%, and 90.66% for the five vulnerabilities.

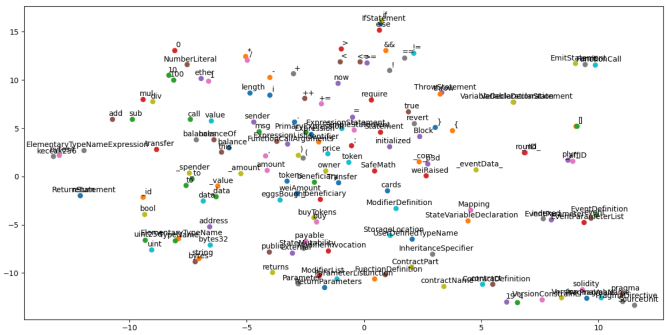

Fig. 11. Visualization of code embedding results.

D. Ablation Study

In order to demonstrate the effectiveness of this code embedding method, we visualized the effect of token embedding using the T-distribution stochastic neighbour embedding (TSNE) algorithm [39]. Dimensions in the range of a few hundreds have been used in the literature with reasonably promising effectiveness [9], [39], [40].

We explore how BiLSTM+ATT performs on single features and which feature poses greater impact on the performance of the method. Therefore, to answer *RQ2*, we carried out ablation experiments to evaluate the performance of the model in single representation and fused representations in terms of *Accuracy*, *Recall*, *Precision*, and *F1-score*.

Result: As shown in Fig. 11, tokens with similar semantic characteristics—such as data types (e.g., unit256, unit, bytes32), operators (e.g., ==, >, >=, +, +=), sensitive keywords (e.g., call, value, msg, sender), and separators (e.g., {, })—are clustered together. This visualization effectively demonstrates the validity of the word embedding representation in capturing meaningful semantic relationships between tokens. In this article, we mainly focus on fusing program slicing feature and AST structure feature of vulnerability code. Table III shows the performance of different code representations (i.e., program slice, AST structure, and their fusion) on the BiLSTM+ATT model. We find that the performance of the fused representation is always *better* than that of single code representations. The expressive power of the model based on program slicing representation is *slightly stronger* than that based on the AST structured representation. Therefore, it can be concluded that

TABLE IV
PERFORMANCE COMPARISON OF WITH THE EXISTING TOOLS/METHODS ON *REENTRANCY*, *TIMESTAMP DEPENDENCE*, *INTEGER OVERFLOW AND UNDERFLOW*, *TX.ORIGIN* AND *UNPROTECTED SELF-DESTRUCT INSTRUCTION*

Methods		Smart Check	Conkas	Honey badger	Mythril	Osiris	Oyente	Securify	Slither	Maian	DR-GCN	TMP	CGE	AME	DeepFusion
Reetrancy	Acc(%)	73.50	55.42	51.40	55.10	43.72	63.39	76.23	78.69	N/A	81.47	84.48	84.49	85.87	**90.84**
	Recall(%)	70.73	24.39	50.02	44.63	32.44	54.88	85.37	80.49	N/A	80.89	82.63	86.21	86.67	**89.49**
	Precision(%)	55.44	37.87	60.10	36.22	40.60	42.06	30.17	58.04	N/A	72.36	78.06	80.38	83.33	**91.84**
	F1(%)	37.42	41.90	51.92	45.38	40.96	42.90	44.59	45.83	N/A	75.94	80.46	83.19	84.97	**90.65**
Timestamp	Acc(%)	70.91	N/A	N/A	54.47	N/A	64.81	N/A	79.89	N/A	77.83	82.75	85.35	86.25	**89.17**
	Recall(%)	55.89	N/A	N/A	51.79	N/A	58.04	N/A	70.71	N/A	75.59	83.82	74.10	76.92	**89.37**
	Precision(%)	50.00	N/A	N/A	42.22	N/A	34.62	N/A	62.34	N/A	70.34	73.24	79.16	78.53	**89.20**
	F1(%)	61.75	N/A	N/A	43.31	N/A	53.04	N/A	72.18	N/A	72.87	78.17	78.02	76.25	**89.28**
Integer Overflow	Acc(%)	68.37	67.69	N/A	62.89	46.85	51.36	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**91.67**
	Recall(%)	37.17	80.53	N/A	11.50	12.39	85.84	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**88.89**
	Precision(%)	65.62	55.49	N/A	61.90	20.90	43.30	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**94.14**
	F1(%)	47.46	65.70	N/A	19.40	15.56	57.57	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**91.43**
Tx.origin	Acc(%)	54.38	N/A	N/A	46.17	N/A	N/A	N/A	70.27	N/A	N/A	N/A	N/A	N/A	**91.56**
	Recall(%)	50.00	N/A	N/A	32.54	N/A	N/A	N/A	61.54	N/A	N/A	N/A	N/A	N/A	**93.65**
	Precision(%)	54.66	N/A	N/A	22.54	N/A	N/A	N/A	60.55	N/A	N/A	N/A	N/A	N/A	**89.74**
	F1(%)	52.01	N/A	N/A	26.63	N/A	N/A	N/A	61.04	N/A	N/A	N/A	N/A	N/A	**91.65**
Selfdestruct	Acc(%)	N/A	N/A	N/A	62.50	N/A	N/A	N/A	65.43	56.63	N/A	N/A	N/A	N/A	**90.48**
	Recall(%)	N/A	N/A	N/A	12.12	N/A	N/A	N/A	42.42	50.78	N/A	N/A	N/A	N/A	**93.75**
	Precision(%)	N/A	N/A	N/A	80.00	N/A	N/A	N/A	60.87	44.36	N/A	N/A	N/A	N/A	**88.24**
	F1(%)	N/A	N/A	N/A	21.05	N/A	N/A	N/A	50	47.35	N/A	N/A	N/A	N/A	**90.31**

the program slicing feature plays a more important role in the performance of *DeepFusion*.

Analysis: The reason why fused representations show better performance than single representation is that the former extract both structure-based and program-based feature information, highlighting vulnerability features while retaining structured information, allowing the model to learn more contractual information at the same time. Next, we analyze the rationality of the better performance of program slice in single representations. The program slice information is extracted in a more fine-grained way. In other words, based on the analysis of smart contract data flow and control flow, *DeepFusion* takes a single variable or statement as the benchmark, extracts the statements affected by the variable or statement, or the statements affecting the variable or statement, and forms code slices. In contrast, due to the peculiarity of the syntax parser, AST structure is extracted from functions. Inside a single function, there may be many variables or statements irrelevant to the vulnerability, which may affect the performance of *DeepFusion* on BiLSTM+ATT and reduce its performance.

Answer to RQ2: Fusion of vulnerability feature fragments and structured information can improve the detection performance of the model. *DeepFusion* improves the accuracy, recall, precision, and F1 score of the one without feature fusion by an average of 3.71%, 3.44%, 5.42%, and 6.30% on the five vulnerabilities.

E. Comparison

To answer *RQ3*, experiments are conducted by comparing the proposed method with nine state-of-the-art smart contract vulnerability detection tools (i.e. SmartCheck, Conkas, Honeybadger, Mythril, Osiris, Oyente, Securify, Slither, and Maian [25], [32], [41]) and four deep learning based methods (i.e. CEG [13], AME [42], TMP, and DR-GCN [21]) based on *accuracy*, *recall*, *precision*, and *F1-score*.

The state-of-the-art smart contract analysis tools are collected via two channels: 1) analysis tools/methods that have been covered by the latest empirical review papers, i.e., [6], [43]; and 2) analysis tools/methods that are available on GitHub. We used the keywords *smart contract security* and *smart contract analysis tools* to search in Github and select the 20 tools with the highest numbers of *stars* from the search results. Next, we selected the nine tools and four methods based on the following criteria:

1) *Criterion 1:* Its input is Solidity source code.
2) *Criterion 2:* It supports command-line interface so that we can apply it to buggy contracts automatically.
3) *Criterion 3:* It is widely used in current research and supports the detection of some of the five targeted vulnerability types.

1) Comparison With Traditional tools/method. Result: Table IV shows the experimental results of the state-of-the-art vulnerability detection tools and methods, where N/A represents that the analysis tool/method is not designed to detect a certain type of vulnerability. Our method shows **better** performance than all the baselines in terms of all the performance metrics.

For most of the vulnerability types, the proposed method shows significant improvement in all the performance metrics compared with the state-of-the-art vulnerability detection tools and methods. The accuracy of *DeepFusion* in detecting the vulnerabilities of *reentrancy*, *timestamp dependence*, *integer overflow and underflow*, *Use tx.origin for authentication*, and *Unprotected Self-Destruct Instruction* is improved by 4.97%, 2.92%, 16.5%, 21.29%, and 25.05%, respectively. *DeepFusion* achieves an average accuracy, recall, precision, and F1 score of 90.74%, 91.03%, 90.63%, and 90.66% for the five vulnerabilities.

Analysis: The existing tools show the following problems in detecting smart contract vulnerabilities: 1) The types of vulnerabilities covered by those detection tools are not

comprehensive enough. Among the nine commonly used detection tools selected, only *Slither* covers the five vulnerability types. 2) The performance of most of these tools is unsatisfactory. Generally speaking, the performance of the analysis tools based on pattern matching or feature capture (represented by *Slither* and *SmartCheck*), is higher than that of the tools based on dynamic analysis (represented by *Mythril*). When we use the analysis tools to detect the contracts, we find that the average detection time of the dynamic analysis is too long due to the path reachability problem, which may generate an unpleasant user experience in actual applications.

2) Comparison With Deep Learning Methods: Here we conduct an in-depth analysis on the deep learning-based methods. *DR-GCN* and *TMP* [21] are based on graph neural networks to detect *reentrancy* and *timestamp dependence* vulnerabilities. They construct contract graphs by extracting control and data flow information from the smart contract code, identifying key function calls or variables as nodes, mapping relationships between functions as edges, and normalizing the resulting graphs. The normalized graphs are finally fed into a graph neural network model for training. Their major difference is that *TMP* takes into account temporal order information when constructing the contract graph while *DR-GCN* does not. *CGE* [13] and *AME* [42] detect *reentrancy* and *timestamp dependence* vulnerabilities based on graph neural networks with added expert rules. They first extract vulnerability features according to expert rules, then use contract graphs to represent the semantic information of source code control flow and data flow, and finally feed the normalized contract graphs into the graph neural network model for training. The main difference between them is that *AME* considers the weight information of different features during model training, while *CGE* does not.

Result: From Table IV, we can see that, although the deep learning methods outperform the existing detection tools, the performance of the former is still worse than our method on most of the metrics.

Analysis: It is found that, although the deep learning methods can use program slicing techniques to retain feature information about vulnerabilities, some valuable information may be lost in the learning process. The reason why *DR-GCN* performs worse than *TMP* is that *DR-GCN* can only use control flow and data flow information to construct the contract graph, while *TMP* also considers time series in addition to the information when constructing the contract graph. The latter connects the nodes in chronological order using directed edges with temporal order, and thus generates a contract graph that contains more feature information. However, *TMP* does not consider the overall structured information. The reason why *TMP* is not as promising as *CGE* and *AME* is that *TMP* only uses program slicing for vulnerability feature extraction, while *CGE* and *AME* use expert rules to extract vulnerability features. These two methods also use directed edges with time sequence to generate contract graphs containing feature information. The reason why *CGE* does not perform as well as *AME* is that *CGE* can only extract vulnerability feature information using expert rules, while *AME* also considers using weight information to highlight the importance of different features. However, none of these methods consider the overall structured information. The coverage of vulnerability types detected by these methods is also not as broad as our approach. While these methods improve performance compared to traditional detection tools, they only use a single representation of information, highlighting vulnerability features while discarding some of the information (e.g., structured information about the contract). Our approach considers both vulnerability features and structured information about the contract, and incorporates an attention mechanism to highlight key features, improving both the coverage and detection accuracy.

Answer to RQ3: *DeepFusion* performs better than the existing methods and detection tools in terms of Accuracy, Recall, Precision, and F1. Our method improves accuracy, recall, precision, and F1 score by an average of 14.15%, 17.30%, 14.14%, and 23.89% compared to the best method across the five vulnerabilities.

F. Efficiency

To answer *RQ4*, we analyze the detection time required by the aforementioned tools/methods.

Result: Table V shows the time used by each tool/method for different vulnerability types, where N/A represents that the analysis tool/method is not designed to detect a certain type of vulnerability. It shows that the time spent by the deep learning-based methods is significantly less than that of the existing detection tools, and the time difference among these detection tools is also remarkable. Deep learning-based methods perform feature extraction through means such as program slicing, in which data preprocessing makes the data more streamlined to highlight vulnerability feature information. As a result, deep learning-based methods are much faster than traditional tools. Among the existing detection tools, *Slither* is the fastest, followed by *SmartCheck*. The remaining seven tools are about the same level.

Analysis: The main reasons for the large differences in the time consumed by these existing detection tools are:

1) The reason why the detection speed of *Slither* and *Smarkcheck* tools is faster is that these two tools are static detection tools so that there is no need to compile and execute smart contracts.
2) The remaining detection tools analyze vulnerabilities at the EVM bytecode level. Therefore, they need to compile smart contracts in advance, resulting in low detection efficiency. Among them, the detection speed of *Mythril* is faster than that of the other EVM bytecode detection tools. *Maian* needs to perform vulnerability detection by deploying smart contracts on private chains and sending transactions, which is relatively time consuming. *Honeybadger* and *Conkas*'s speed is unsatisfactory since their detection mechanisms need to consider the internal information of a contract.

The reason why the training and testing time of *DeepFusion* is longer than that of the other four methods is that *DeepFusion* is

TABLE V
EXECUTION TIME (UNIT: SECONDS) OF DIFFERENT TOOLS FOR THE FIVE VULNERABILITY TYPES

	Slither	Osiris	Oyente	Smart Check	Honey badger	Securify	Mythril	Conkas	Maian	DR-GCN		TMP		CGE		AME		DeepFusion	
										train	test	train	test	train	test	train	test	train	test
Reetrancy	1382	29753	15875	5625	59457	22989	40623	61952	N/A	**104.51**	**0.21**	121.02	0.28	129.47	0.32	131.36	0.35	191.20	0.46
Timestamp	905	N/A	13024	2104	N/A	N/A	20108	N/A	N/A	**122.13**	**0.23**	143.93	0.25	151.64	0.34	155.79	0.38	268.68	0.64
Tx.origin	326	N/A	N/A	1864	N/A	N/A	13326	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**141.05**	**0.51**
Integer Overflow	N/A	16372	3805	1273	N/A	13689	11719	34856	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**132.41**	**0.36**
Selfdestruct	173	N/A	N/A	N/A	N/A	N/A	7875	N/A	25596	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**122.21**	**0.29**
Average Time	660	23062.5	10901.33	2716.5	59457	18339	18730.2	48404	25596	**113.32**	**0.22**	132.18	0.27	140.56	0.33	143.58	0.37	171.11	0.45

trained and tested on fused data representations, which is more time-consuming for data processing in comparison to the two existing methods based on single data representations. *TMP* and *DR-GCN* consume less time than *CEG* and *AME*, since the former is relatively simple. In contrast, *CEG* and *AME* consider using expert rules for more fine-grained feature extraction, so they take relatively more time. All of these methods focus on feature extraction at the source code level, and none of them consider the overall structured information of smart contracts. However, the time differences are relatively minor and the overall performance of *DeepFusion* is still far better than that of the other four deep learning methods. Compared with the nine detection tools, the proposed detection method based on deep learning is more conducive to the detection of smart contract vulnerabilities. In addition to improved detection time and performance, the proposed method does not rely on manual detection rules, which can better adapt to the evolution and diversification of smart contract vulnerabilities.

Answer to RQ4: Although the efficiency of *DeepFusion* is not as high as the detection method of single representation information, the gap is not very large, and the efficiency is significantly better than that of traditional detection tools. *DeepFusion* consumes much less time (average 172.56 s compared to the fastest traditional tools (average 660 s).

G. Threats to Validity

We recognize the following threats to the validity of our research.

Internal validity: The potential threats to internal validity originate from the following three aspects:

1) *Restricted Extensibility:* So far we can employ *DeepFusion* to detect five types of smart contract vulnerabilities. The implementation of *DeepFusion* relies on researchers' in-depth understanding of the smart contract vulnerabilities. When constructing and labeling datasets, we use manual verification to judge whether a contract to be tested contains some types of vulnerabilities. Such a process may be inaccurate due to subjectivity. To alleviate this threat, we use a combination of tool-based detection and manual verification (detailed in Section III-B). The ground truth of the dataset is obtained by achieving the consensus between two authors who conducted independent labeling. In addition, we publicize some datasets for other researchers and developers to conduct verification and reviews.
2) *Limited Types of Detection Vulnerabilities: DeepFusion* is only able to detect five types of vulnerabilities. According to the study, as the number of Ethereum smart contracts grows and the deployment environment changes, over five types of vulnerabilities have emerged [28]. However, most vulnerability types lack a fair and open dataset for performing the research in this area. Therefore, we constructed a dataset containing the five types of vulnerabilities. In selecting these types of vulnerabilities, we analyzed reported security incidents and conducted a literature review to ensure our dataset focuses on the most harmful and frequent vulnerabilities. This approach not only enhances the relevance of our research findings but also lays the foundation for comprehensively covering major threats in this field. We will expand the scope of vulnerability detection to other types in future research work.
3) *The Types of Contracts are Limited: DeepFusion* is designed for vulnerability detection in smart contracts written in Solidity, which does not encompass all programming languages used for smart contracts. However, since Solidity is the preferred language of Ethereum developers and the majority of smart contracts are developed using Solidity, this significantly mitigates the limitation of the contract types covered.

External Validity: The potential threat to external validity results from the tools that *DeepFusion* relies on. *DeepFusion* relies on three open-source tools (i.e., *solc*, *Slither,* and *ANTLR*) to construct a contract's control and data flows. The performance and reliability of these tools directly influence the validity of *DeepFusion*'s results. To mitigate this threat, it is important to note that *solc*, *Slither*, and *ANTLR* are established tools in the domain of smart contract analysis and parsing. These tools are maintained by dedicated organizations and developers, ensuring regular updates and improvements. Their widespread adoption in the industry further validates their reliability and effectiveness, thereby reducing the impact of this threat.

V. RELATED WORK

According to the underlying technique, smart contract vulnerability detection can be classified into the following categories: traditional methods and deep learning-based methods.

A. Traditional Methods

Traditional tools for detecting smart contract vulnerabilities are categorized into three types: abstract interpretation, symbolic

execution, and fuzzing. Feist [35] proposed *Slither* based on symbolic execution. *Slither* initially transforms Solidity code into an intermediate representation named SlithIR, and then utilizes the preserved semantic information for vulnerability detection. Tsankov et al. [44] proposed *Securify* based on abstract interpretation, compared with the vulnerability standards of SWC and *Slither*, it defines contracts and violation patterns by analyzing the dependence graph of the program, so as to extract the semantic information of vulnerabilities. Tikhomirov et al. [32] proposed *SmartCheck*, which finds contract vulnerabilities by converting Solidity source code into an XML-based intermediate representation and comparing it to a predefined XPath path. Luu et al. [45] proposed *Oyente*, which constructs a contract control flow graph at the bytecode level. Torres et al. [46] proposed *Osiris*, which generates basic blocks of contracts containing integer overflow errors by analyzing CFGs constructed at the bytecode level. Consensys [47] proposed *Mythril*, which simulates contract invocations through multiple symbolic executions for vulnerability detection. Fu et al. [48] proposed *EVMFuzzer* to discover vulnerabilities in EVMs. Seed contracts are generated and respectively sent to the target EVM and the benchmark EVM. The discrepancies between their execution results are eventually analyzed. Kalra et al. [49] proposed a framework, *ZEUS*, to test the validity of smart contracts. *ZEUS* analyzes the security of contracts through symbolic execution and abstract interpretation. Mossberg et al. [50] proposed *Manticore*, a framework for dynamic symbolic execution through analysis of binaries and smart contracts. Cook et al. [51] proposed a smart contract attack analysis tool called *DappGuard* that identifies attacks by analyzing transaction logs. So et al. [52] proposed a security analyzer called *VeriSmart* to verify Ethernet smart contracts that can be used to detect integer overflow and underflow vulnerabilities. Later, So et al. [53] devised a symbolic execution technique called *SmarTest* to find vulnerable transaction sequences in smart contracts.

To sum up, the traditional smart contract vulnerability detection mainly relies on human experts to define detection rules. and have the following disadvantages: poor scalability, low accuracy, and high cost.

B. Deep Learning-Based Methods

Smart contract vulnerability detection based on deep learning mainly uses deep learning to analyze the lexical, syntax, control flow, data flow, and other information of the code. Related research [18] shows that it has higher accuracy and completeness than traditional detection. Qian et al. [54] proposed Rechecker, the first model to use deep learning for automated smart contract *Reentrancy* vulnerability detection. Zhuang et al. [21] proposed TMP and DR-GCN, which no longer perform feature "squashing" but instead use graph neural networks for vulnerability feature learning by constructing graphs in the form of nodes and edges, supporting three vulnerabilities. Liu et al. [13] proposed CGE that combines graph neural networks and traditional expert mode to extract contract vulnerability features. It supports three vulnerabilities. Liu et al. [42] designed AME to incorporate expert rules into graph neural networks. The difference from CEG is that AME can obtain the distribution map for each feature weight. This method supports the detection of three types of smart contract vulnerabilities. Wang et al. [4] developed Contractward, a machine learning-based model capable of detecting six types of smart contract vulnerabilities. Jie et al. [55] developed a method for smart contract vulnerability detection. By combining the static feature information of the extracted source code layer and the bytecode layer, and feeding it into a neural network model for training, this method can only detect if a test contract is vulnerable other than identifying specific types of vulnerabilities. Cai et al. [56] introduced a novel approach that enhances smart contract detection capabilities by integrating slicing union graphs with graph neural networks. Jiang et al. [57] developed VDDL, a deep learning-based smart contract vulnerability detection model that integrates various deep learning strategies to understand and predict potential security vulnerabilities in smart contracts. Zeng et al. [58] introduced SolGPT, a static vulnerability detection model leveraging the GPT architecture, to improve the security of smart contracts. However, it does not provide the classification detection function. Mi et al. [59] introduced VSCL, a method that decompiles bytecode into operational sequence code. Using control flow graphs and depth-first search algorithms, it generates a new sequence and employs deep neural networks for the analysis and detection of vulnerabilities. Meanwhile, Huang et al. [60] presented a multitask learning-based model for detecting vulnerabilities in smart contracts. This model relies on a multiheaded attention mechanism network to learn and extract feature vectors from contracts, and then uses convolutional neural networks to detect and identify vulnerabilities. It supports the detection of arithmetic vulnerability, reentrancy vulnerability and contract contains unknown address vulnerability.

This type of methods suffers from the following limitations: although the existing deep learning detection methods are better than traditional tools in performance, the single representation information will prevent the model from learning complete feature information, and the coverage of existing vulnerability types needs to be improved.

VI. Conclusion and Future Work

In this article, we attempt to explore smart contract vulnerability detection methods that fuse program slicing information with AST structured information and feed it into a deep learning model for training. We obtained positive findings that the fused data improves the interpretability of the data while preserving the smart contract vulnerability features. Extensive experimental results show that *DeepFusion* significantly outperforms traditional vulnerability detection tools and state-of-the-art deep learning methods. We believe that *DeepFusion* is an important step forward in the interpretability and accuracy of smart contract data based on deep learning.

In future work, we will collect more real-world smart contracts and build datasets covering a wider range of vulnerability detection. In addition, we will improve the existing BiLSTM model to enhance its performance. We will also explore the possibility of incorporating other fine-grained information into *DeepFusion*. Furthermore, we consider extending the types of smart contracts from additional platforms into *DeepFusion*.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Bus. Rev.*, vol. 4, no. 2, p. 15, 2008.

[2] V. Buterin et al., "A next-generation smart contract and decentralized application platform," *White Paper*, vol. 3, no. 37, pp. 2–1, 2014.

[3] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 259–269, 2018.

[4] W. Wang et al., "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, Apr.–Jun. 2021.

[5] W. Zou et al., "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2019.

[6] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang, "Smart contract security: A practitioners' perspective," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1410–1422.

[7] S. Singh, A. S. Hosen, and B. Yoon, "Blockchain security attacks, challenges, and solutions for the future distributed IoT network," *IEEE Access*, vol. 9, pp. 13938–13959, 2021.

[8] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–11.

[9] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2874–2891, Dec. 2020.

[10] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, "Aroc: An automatic repair framework for on-chain smart contracts," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4611–4629, Nov. 2022.

[11] T. Chen et al., "Towards saving money in using smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., New Ideas Emerg. Technol. Results*, 2018, pp. 81–84.

[12] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022.

[13] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 2, pp. 1296–1310, Feb. 2021.

[14] H. Wu et al., "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 378–389.

[15] X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, "DeeSCVHunter: A deep learning-based framework for smart contract vulnerability detection," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2021, pp. 1–8.

[16] M. Zhang, P. Zhang, X. Luo, and F. Xiao, "Source code obfuscation for smart contracts," in *Proc. 27th Asia-Pacific Softw. Eng. Conf.*, 2020, pp. 513–514.

[17] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2018, pp. 1–15.

[18] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, Sep./Oct., 2019.

[19] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *J. Netw. Comput. Appl.*, vol. 177, 2021, Art. no. 102857.

[20] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? Investigating the selfdestruct function on ethereum," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 1–37, 2021.

[21] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network.," in *Proc. Int. Joint Conf. Artif. Intell.*, 2020, pp. 3283–3290.

[22] W. Dingman et al., "Classification of smart contract bugs using the NIST bugs framework," in *Proc. IEEE 17th Int. Conf. Softw. Eng. Res., Manage. Appl.*, 2019, pp. 116–123.

[23] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Proc. Int. Conf. Princ. Secur. Trust*, 2017, pp. 164–186.

[24] J. Liang and Y. Zhai, "Scgru: A model for Ethereum smart contract vulnerability detection combining CNN and Bigru-attention," in *Proc. IEEE 8th Int. Conf. Signal Image Process.*, 2023, pp. 831–837.

[25] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 530–541.

[26] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, "A survey on smart contract vulnerabilities: Data sources, detection and repair," *Inf. Softw. Technol.*, vol. 159, 2023, Art. no. 107221.

[27] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, 2020.

[28] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in Ethereum smart contracts," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 139–150.

[29] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., Companion*, 2018, pp. 65–68.

[30] B. Wang, H. Chu, P. Zhang, and H. Dong, "Smart contract vulnerability detection using code representation fusion," in *Proc. 28th Asia-Pacific Softw. Eng. Conf.*, 2021, pp. 564–565.

[31] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, Springer, 2016, pp. 79–94.

[32] S. Tikhomirov et al., "Smartcheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.

[33] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: Issues, techniques, and future challenges," *Empirical Softw. Eng.*, vol. 26, no. 6, pp. 1–44, 2021.

[34] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24416–24427, 2020.

[35] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.

[36] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[37] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 783–794.

[38] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," 2016, *arXiv:1608.02715*.

[39] C. Gao, J. Zeng, X. Xia, D. Lo, M. R. Lyu, and I. King, "Automating app review response generation," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 163–175.

[40] J. Chen et al., "Defectchecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022.

[41] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.

[42] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in *Proc. 30th Int. Joint Conf. Artif. Intell.*, Montreal, Canada, Aug. 2021, pp. 2751–2759.

[43] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proc. 28th Annu. Int. Conf. Comput. Sci. Softw. Eng. (CASCON)*, Markham, ON, Canada, Oct. 2018, pp. 103–113.

[44] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.

[45] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[46] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.

[47] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, vol. 9, 2018, Art. no. 54.

[48] Y. Fu et al., "EVMFuzzer: Detect EVM vulnerabilities via fuzz testing," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 1110–1114.

[49] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. Ndss*, 2018, pp. 1–12.

[50] M. Mossberg et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 1186–1189.

[51] T. Cook, A. Latham, and J. H. Lee, "Dappguard: Active monitoring and defense for solidity smart contracts," *Retrieved Jul.*, vol. 18, 2017, Art. no. 2018.
[52] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1678–1694.
[53] S. So, S. Hong, and H. Oh, "{SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur. 21)*, 2021, pp. 1361–1378.
[54] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19685–19695, 2020.
[55] W. Jie, A. S. V. Koe, P. Huang, and S. Zhang, "Full-stack hierarchical fusion of static features for smart contracts vulnerability detection," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, 2021, pp. 95–102.
[56] J. Cai, B. Li, J. Zhang, X. Sun, and B. Chen, "Combine sliced joint graph with graph neural networks for smart contract vulnerability detection," *J. Syst. Softw.*, vol. 195, 2023, Art. no. 111550.
[57] F. Jiang et al., "VDDL: A deep learning-based vulnerability detection model for smart contracts," in *Proc. Int. Conf. Mach. Learn. Cyber Secur.*, Springer, 2022, pp. 72–86.
[58] S. Zeng, H. Zhang, J. Wang, and K. Shi, "SOLGPT: A GPT-based static vulnerability detection model for enhancing smart contract security," in *Proc. Int. Conf. Algorithms Architectures Parallel Process.*, Springer, 2023, pp. 42–62.
[59] F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, and L. Khan, "VSCL: Automating vulnerability detection in smart contracts with deep learning," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency*, 2021, pp. 1–9.
[60] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart contract vulnerability detection model based on multi-task learning," *Sensors*, vol. 22, no. 5, 2022, Art. no. 1829.

Hanting Chu received the M.S. degree in Internet of Things engineering from Bohai University, Jinzhou, China, in 2020. She is currently working toward the Ph.D. degree in computer science and technology with the College of Computer and Information, Hohai University, Nanjing, China.

Her current research interests include smart contract security and software engineering.

Pengcheng Zhang (Member, IEEE) received the Ph.D. degree in computer science from Southeast University, Nanjing, China, in 2010.

He is currently a Full Professor with the College of Computer and Information, Hohai University, Nanjing, China, and was a Visiting Scholar with San Jose State University, USA. His research interests include software engineering, service computing and data science. He has published research papers in premiere or famous computer science journals, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, IEEE TRANSACTIONS ON BIG DATA, IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, IEEE TRANSACTIONS ON CLOUD COMPUTING and IEEE TRANSACTIONS ON RELIABILITY. He was the Co-chair of IEEE AI Testing 2019 conference. He served as a technical program committee member on various international conferences.

Hai Dong (Senior Member, IEEE) received the Ph.D. degree in computer science from Curtin University, Perth, Australia.

He is currently a Senior Lecturer with the School of Computing Technologies, RMIT University, Melbourne, Australia. He was previously a Vice-Chancellor's Research Fellow in RMIT University and a Curtin Research Fellow in Curtin University. His primary research interests include: services computing, edge computing, blockchain, cyber security, machine learning and data science. His publications appear in *ACM Computing Surveys*, IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, etc.

Yan Xiao received the Ph.D. degree in computer science from the City University of Hong Kong, Hong Kong, in 2019.

She is a Research Fellow with the School of Computing at National University of Singapore. Her research focuses on trustworthiness of deep learning system and AI applications in software engineering.

Shunhui Ji received the B.S. degree in computer science and technology and the Ph.D. degree in computer software and theory from Southeast University, Nanjing, China, in 2008 and 2015, respectively.

She is currently an Associate Professor with the College of Computer and Information, Hohai University, Nanjing, China. Her research interests include service computing, cloud computing, software modeling, analysis, testing, and verification. She is a Reviewer of some international conferences and journals.