DeepFusion: Smart contract vulnerability detection via deep learning and data fusion

Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji

ABSTRACT

With hundreds of millions of dollars of transactions executed by smart contracts every day, the issue of smart contract security has gained a lot of attention over the past few years. Traditional vulnerability detection methods rely heavily 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. In addition, the number of available vulnerability datasets is also insufficient. To address these limitations, in this paper, we propose *DeepFusion*, a new method to combine structured information from Abstract Syntax Tree (AST) with program slicing information. Specifically, we have developed automated tools to extract feature information from the source code, then convert the source code into an AST to extract the structured information. Thereafter, the code features and global structured features are fused into data and the fused data is then fed into the Bi-directional Long Short-Term Memory+Attention (BLSTM+ATT) model to obtain vulnerability detection results. 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 *re-entrancy*, *timestamp dependence*, *integer overflow and underflow*, *Use tx.origin for authentication*, and *Unprotected Selfdestruct Instruction*.

Index Terms—Smart Contract, Vulnerability Detection, Deep Learning, AST, Program Slicing

IEEE Reference Format:
H. Chu, P. Zhang, H. Dong, Y. Xiao and S. Ji, "DeepFusion: Smart Contract Vulnerability Detection Via Deep Learning and Data Fusion," in IEEE Transactions on Reliability, doi: 10.1109/TR.2024.3480010.

1 INTRODUCTION

Blockchain is essentially a distributed shared transaction ledger, the concept of which was first proposed by Nakamoto in 2008 [29]. Blockchain technology is decentralized, tamper-proof and traceable. Smart contract, one of the most successful applications of blockchain technology, has captured the attention of academia and industry like[2][22][40]. A smart contract is a computer program that runs on a blockchain platform, usually written in Solidity, a Turing-complete high-level programming language[50]. 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[38] incident occurred in June 2016, attackers exploited a re-entrancy 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[34].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 being 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.

Smart contract vulnerability detection is an important topic in smart contract security research. Many tools have been developed for detecting vulnerabilities[5, 8–10, 16, 18–20, 23, 26, 33, 42–44, 46]. 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 [24, 49]. However, such manually developed detection rules may lead to high false positive rates and are unable to cope with complex vulnerability patterns[21]. 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[4].

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 [49]. 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* [48], 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 pre-defined 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 paper are:

(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 selfdestruct instruction* vulnerability. To this end, we propose a simple and effective feature fusion based detection network model based on BLSTM+ATT. To the best of our knowledge, this is the first time such 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*.

The rest of this paper is organized as follows: Section 2 introduces background information in relation to this research, including smart contracts vulnerability detection, vulnerability classification framework, vulnerability type and call graph. Section 3 presents our approach. In Section 4, we use a collected buggy smart contract dataset to validate our method. After discussing the related work in Section 5, we conclude the paper and plan future work in Section 6.

2 BACKGROUND KNOWLEDGE

2.1 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[5]. Therefore, we select the following five vulnerabilities as the research targets.

Re-entrancy. The *re-entrancy* vulnerability is one of the most devastating types of security vulnerabilities, which was exploited in the famous DAO attack[25, 39]. This vulnerability typically occurs when a transfer is made to an external user address. It is triggered by the external user address recursively calling the same function of the contract[21]. The default smart contract in the Solidity language contains a callback function without a function name and parameters, which is automatically triggered when a transfer is received. When the smart contract triggers a transfer operation, a *re-entrancy* attack can occur before the operation modifies the contract state variable. Consequently, an attacker can trigger a malicious fallback function that causes the smart contract to call the transfer function repeatedly until the account balance reaches 0.

As shown in Figure 1, a withdraw function in *contract 1* is to perform the rollback function. The attacker initiates a transaction by attacking the contract by repeatedly calling the withdraw function in contract 1 until the victim's account balance is 0 or the gas is exhausted.

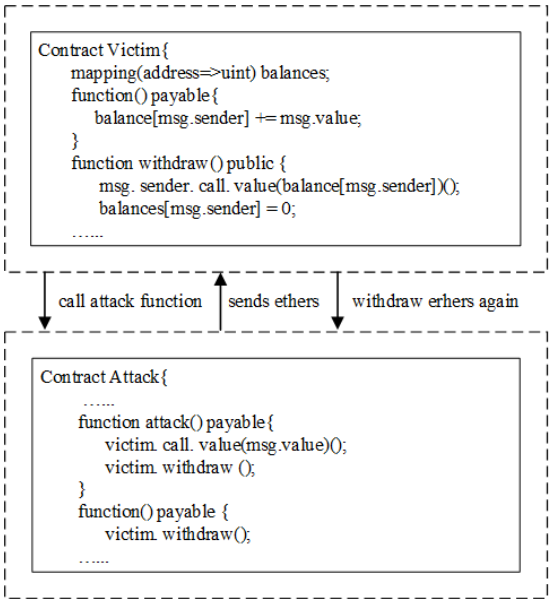

Figure 1: Re-entrancy

Timestamp Dependency. The Ethereum protocol stipulates that miners can freely set the timestamp of the block when the timestamp difference is less than 900 seconds[14]. 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 dependency 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[12], 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 that smart contracts are exploited by attackers to launch attacks due to integer overflow and underflow vulnerabilities, which is a high-risk vulnerability[35]. An attacker may use this vulnerability to transfer a large amount 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[7]. 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→B→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 invoking the contract.

As shown in Figure 2, an attacker can deploy an attack contract, including a fallback function that calls the *sendTo* function in *MyContract*. Since *tx.origin == owner* is used as the judgment condition in the *MyContract* contract, *tx.origin* represents the address from which the transaction was initially sent. By attacking the contract in the form of disguising the identity, the attacker can obtain the transfer qualification, thereby causing ether theft.

```
contract MyContract {
    address owner;
    function MyContract() public {
        owner = msg.sender;
    }
    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        receiver.transfer(amount);
    }
}
```

Figure 2: Use *tx.origin* for authentication

Unprotected Selfdestruct Instruction. Since smart contracts cannot be deleted once being deployed, they provide a destruction function that requires the user to introduce a suicide function when developing a contract[33]. Once a problem is found in the contact, property loss can be avoided by calling the suicide function to destroy the contract and transferring the ethers to a specified address. However, if a smart contract containing a suicide function lacks permission control, then any user can call the suicide function to kill the contract and transfer the ethers of the contract to their specified address. This error can affect the function of the contract and cause financial loss.

2.2 Vulnerability Classification Framework

Many studies on smart contract vulnerability classification and standards have been conducted[13]. In 2017, Atzei et al.[1] analysed the security vulnerabilities of Ethereum smart contracts. For the first time, they classified smart contract vulnerabilities into 3 levels: programming language, virtual machine, and blockchain. In the same year, Dika et al.[14] followed Atzei et al.'s classification and categorised known security issues with the security levels of low, medium and high risk respectively. In 2018, the Decentralized Application Security Project (DASP) has summarized 10 types of high-risk smart contract vulnerabilities. In 2020, Chen et al.[3] collected real smart contracts from Ether and posts, and defined 20 smart contract flaws in terms of potential security, usability, maintainability and reusability. Zhang et al.[47] proposes a comprehensive vulnerability classification framework called *JiuZhou* by extending the *IEEE Standard Classification for Software Anomalies*, which summarizes 49 types of vulnerabilities and their severity levels. In this paper, we focus on the deep learning detection of five kinds of smart contract vulnerabilities with high severity.

2.3 Call Graph

A call graph (also known as call multigraph)[15] is a control flow graph that represents the call relationship between subprograms of a computer program. A simple application of call graph is to find procedures that have never been called. Call graphs can be used for 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 change impact prediction. Call graphs can also be used to detect program execution exceptions or code injection attacks. In this paper, we analyze the calling relationship between internal functions of a contact based on the call graph generated by *Slither*, so as to obtain the possible data flow and control flow information.

3 OUR METHOD

3.1 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 5 types of severe vulnerabilities. We use Figure 3 to represent the whole 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 3.2. Second, for different vulnerability types, *DeepFusion* extracts the corresponding target function fragments (described in Section 3.3: *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 3.4) and AST structured information (obtained by *AST Structured Information Extractor* introduced in Section 3.5). *DeepFusion* fuses the two kinds of information, and then enters the information into a pre-designed model for training and prediction (implemented by *Model Training and Model Prediction* depicted in Section 3.6).

3.2 Data Collection

We briefly describe the process of data collection before formally introducing the proposed vulnerability detection approach. The source of the dataset is twofold: First, we used the keywords of *smart contract vulnerability, vulnerable smart contracts, buggy smart contracts*, and *smart contracts defects* to search smart contracts on GitHub and Gitter chat room[1]; Second, we used *Karl*[2] to collect new smart contracts from the Ethereum blockchain website[3], where *Karl* is a tool used together with *Mythril* for real-time blockchain monitoring. They can provide addresses of smart contracts that may cause vulnerabilities. For data labelling, we adopted *Mythril*, *Slither*, *Oyente* and *SmartCheck* to detect the collected contracts. When 3/4 of the tools report that there is a problem with a certain line of a contract, we conduct a manual verification and label the vulnerability once it is verified. We then add the labelled contract into the vulnerability dataset.

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

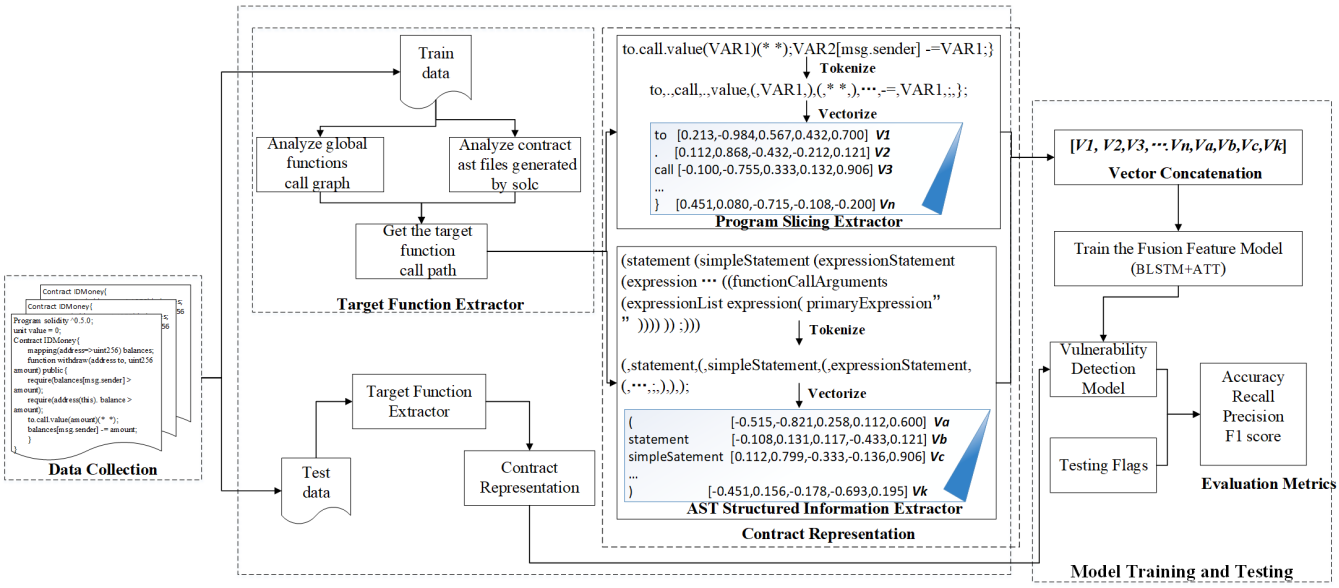

Figure 3: Method Flowchart.

3.3 Target Function Extractor

The goal of Target Function Extractor is to obtain function fragments related to vulnerabilities. *DeepFusion* analyzes the global function-call graph 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. In this paper, we use two representative vulnerabilities, *re-entrancy* and *timestamp dependence*, as examples to describe in detail how Target Function Extractor works. We briefly introduce Target Function Extractor(s) of the remaining 3 types of vulnerabilities in Table 1.

3.3.1 *Target Function Extractor of Re-entrancy (TFER).*

Re-entrancy 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, for vulnerable contracts, *TFER* analyzes the abstract syntax tree json file generated by *solc* (official compiler of Solidity) to locate the function path containing the *call-statement*. For non-vulnerable contracts, *TFER* also extracts function paths containing *call-statements* (A contract containing *call.value* does not necessarily contain *re-entrancy*. For example, when the decrement of the ledger variable happens before the calling statement then the contract usually does not suffer from *re-entrancy* vulnerability). Additionally, for the objectivity of the dataset, the deposit paths in which a function is declared as *payable* will be extracted for the extraction of non-vulnerable function fragments. *TFER* then extracts the corresponding complete functional fragments according to the function paths.

3.3.2 *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. For vulnerable contracts, *TFET* extracts function paths containing *block.timestamp* or *now* in the collected dataset. For non-vulnerable contracts, *TFET* also extracts function paths containing *block.timestamp* or *now* (A contract containing *block.timestamp* does not necessarily contain *timestamp dependence*. For example, if *block.timestamp* is assigned to a variable, which is not used or restricted by strict condition statements e.g. *require* or *assert* in the next process. We usually think that this contract does not contain timestamp dependency). *TFET* then extracts the corresponding complete functional fragment according to the function paths.

Algorithm 1: Program Slicing Extractor Algorithm

Input: Target functional fragments' source code SC and vulnerability standard VS
Output: Vulnerability program slices P

```
ContractAstSet C = getContractASTBySolc(SC);
VariableSet VC = getSpecificVariable(SC) according to VS;
foreach contractAst in C do
    if contractAst has data dependence with VC then
        statementSet dd = getSourceCodeStatement(contractAst);
        p append dd
    end
    if contractAst has control dependence with VC then
        statementSet cd = getSourceCodeStatement(contractAst);
        p append cd
    end
end
return P;
```

3.4 Program Slicing Extractor

Program slicing techniques[41] are applied to extract the semantic information of the code. Program Slicing Extractor is designed

Table 1: The *Target Function Extractor(s)* and *Program Slicing Extractor(s)* of the remaining 3 types of vulnerabilities

Bug type	Target Function Extractor	Program Slicing Extractor
Tx.origin	***DeepFusion*** first searches for function paths containing the *tx.origin-statements* in contracts. For vulnerable contracts, for example, if *tx.origin-statements* exist in function *modifier*, all function fragments modified by this function *modifier* become our target functions. For non-vulnerable contracts, if *msg.sender==tx.origin* is used to reject an external contract to invoke the current contract, we will identify this function as the objective function.	For vulnerable contracts, for example, if *tx.origin-statements* exist in the *require-statement*, ***DeepFusion*** extracts all the following statements from the target function. For non-vulnerable contracts, if *tx.origin* is used as a variable of *address* type, ***DeepFusion*** obtains all statements that operate with this *address* variable, and then forms these statements into slices.
Integer Overflow	***DeepFusion*** first searches for function paths containing the *arithmetic operations-statements* in contracts. For vulnerable contracts, for example, if all function fragments that have *arithmetic operations* are not bound by the safemath library, or have no conditional statements to constrain, we mark the function that has *arithmetic operations* as the target function. For non-vulnerable contracts, if there is a function with *arithmetic operations* subject to the above conditions, we will identify this function as the objective function.	For vulnerable contracts, for example, if *arithmetic operations* functions exist and not subject to conditions, ***DeepFusion*** extracts all statement objective functions. For non-vulnerable contracts, if *arithmetic operations* functions have conditional constraints, ***DeepFusion*** slices these statements.
Selfdestruct	***DeepFusion*** first searches for paths containing a *suicide-statement* or a *selfdestruct-statement*. For vulnerable contracts, for example, if there is a *suicide-statement* or *selfdestruct-statement*, but it is not bound by a conditional statement and can be called by anyone, we mark the function as the target function with a *suicide-statement* or a *selfdestruct-statement*. For non-vulnerable contracts, if there is a *suicide-statement* or a *selfdestruct-statement* subject to a strict conditional statement, we will identify this function as the target function.	For vulnerable contracts, for example, if *suicide-statement* or *selfdestruct-statement* is present and unconditional, ***DeepFusion*** extracts all statements in the target function. For non-vulnerable contracts, if the *suicide-statement* or *selfdestruct-statement* has conditional constraints, ***DeepFusion*** slices these statements.

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. Like Section 3.3, we also select two vulnerability types, *re-entrancy* and *timestamp dependence* as examples. The rest of the vulnerability types are shown in Table 1.

3.4.1 *Program Slicing Extractor of Re-entrancy (PSER).*

For *re-entrancy* vulnerability, *PSER* first extracts the *address* variable in front of the *call-statements* based on the corresponding target functional fragments in Section 3.3.1, and all the statements that operate on the *address* variable to form program slices.

3.4.2 *Program Slicing Extractor of Timestamp Dependence(PSET).*

For *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, and then composes the above statements into slices.

3.5 AST Structured Information Extractor

AST Structured Information Extractor aims at obtaining the structured information of the functional fragments in Section 3.3. We adopt ANTLR[4] (AN-other Tool for Language Recognition) to parse those function fragments into AST (Abstract Syntax Tree)[45]. ANTLR is a powerful parser generator for translating structured text and analysing language. AST is a tree-like representation of abstract syntax structure of source code. Here, we deploy depth-first search to serialize the AST of the function fragments. It should be noted that, for different types of vulnerability fragments in Section 3.3, we use the same traversal method to convert them into serialized AST fragments.

3.6 Model Training and Model Prediction

3.6.1 *Normalization and Embedding.*

Before inputting the data of Section 3.4 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. Punctuation without actual semantic information (i.e., ";", ",") will be eliminated. We need to use word segmentation to convert the program slicing of Section 3.4 and the AST structured fragments of Section 3.5 into code tokens, and then use word embedding to

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

convert the above tokens into vectors, which are ready to be inputted into the model for training and testing. It should be noted that the dimension we choose in this paper to embed words here is 150. Dimensions in the range of a few hundreds have been used in the literature with reasonably good effectiveness[6, 17, 18]. We will deepen the discussion on this dimension as future work.

3.6.2 *Our model.*

The work of Dam et al.[11] 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. BLSTM).

The Long Short-Term Memory (LSTM) model, designed to alleviate the RNNs gradient disappearance problem, was devised to more accurately find and exploit long-range context using special memory cells. An LSTM layer consists of a number of recurrently connected memory blocks. Each block contains one or more recurrently connected memory cells C_t and three multiplicative units, namely the input i_t, output o_t, and forget gate f_t, which control the information flow inside the memory block. The forget gate decides what information to be discarded from the cell state. The input gate determines how much new information to be added to the cell state. The output gate decides what value to be generated as output. The hidden state h_t for an LSTM cell can be calculated by Eq.1-6, where $\tilde{C}_t$ is a vector of new candidate values, σ is sigmoid function, tanh is hyperbolic tangent function, and $*$ represents matrix multiplication and element-wise product. We employed a bi-directional LSTM (BLSTM) model to obtain both past and future bi-directional information of sequences, where the sequence features fed into the BLSTM model are multi-dimensional. 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.

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

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

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

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

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

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

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

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

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

Where Eq.7 is the formula for calculating the score of each feature vector. After capturing the score of each feature vector, Eq.8 obtains the normalised weights for the attention mechanism by performing a normalisation operation using the softmax function. T indicates the final output.

3.6.3 *Training and Prediction.*

In this paper, we intend to integrate two features containing vulnerability information. One is the program slicing feature of vulnerabilities, and the other is the structural feature of AST obtained by serializing an AST after converting a vulnerability fragment into the AST. Therefore, for model training, we concatenate these two word vectors as a new fusion vector described in Section 3.4 and Section 3.5. The new vector is a code representation combining these two sets of feature information. Finally, the concatenated vector is utilized as the input to train a deep learning based vulnerability detection model. We build five types of RNN models[32] and Attention. 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, the source code will be handled by aforementioned steps(i.e., Target 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.

4 EXPERIMENT AND EVALUATION

Dataset. We manually collected and constructed a dataset[5] 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, 4,313 smart contracts containing *re-entrancy* vulnerability, *timestamp dependence* vulnerability, *tx.origin* vulnerability, *Integer Overflow and Underflow* and *Unprotected Selfdestruct Instruction* are selected. We used this dataset to evaluate the proposed method. 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 percent of the dataset into a training set and the remaining 20 percent into a test set. We repeated each experiment 10 times and calculated the average values to obtain the final experimental results.

We have attempted to respond to the following four questions:

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

4.1 Performance

To answer *RQ1*, we conduct experiments for detecting the five types of vulnerabilities. Followed by Section 3.6.2, we use LSTM, GRU, BLSTM, BLSTM+ATT and RNN, for feature extraction and training in code. 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

[5]https://anonymous.4open.science/r/DeepFusion-FC47

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]. Through the experiments, we finally adopted learning rate = 0.002, dropout rate = 0.4, batch size = 64, and the dimension of each token vector = 100 for all the candidate models.

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 2 and 3. In terms of processing long sequence data, the performance of the LSTM model is better than that of the traditional RNN and GRU model. Therefore, LSTM is more suitable than RNN and GRU for processing smart contract data. BLSTM can simultaneously capture bidirectional (i.e. forward and backward) long-term and short-term dependencies, which shows better performance than LSTM. Further, attention mechanism can highlight key features and greatly improve the performance of the BLSTM model. As a result, BLSTM+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 BLSTM+ATT.

4.2 Ablation Study

In this paper, we mainly focus on fusing program slicing feature and AST structure feature of vulnerability code. We rely on BLSTM+ATT to explore how the model 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. Table 4 shows the performance of different code representations (i.e., program slice, AST structure and their fusion) on the BLSTM+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 the program slicing feature plays a more important role in the performance of *DeepFusion*.

Analysis. The reason why fused representation shows better performance than that of single representations is that the former is more capable of capturing the features of vulnerabilities than the latter in terms of both structure and program based feature extraction and learning (detailed in Section 3.6.2).

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 BLSTM+ATT and reduce its performance.

Answer to RQ2: Fusion of vulnerability feature fragments and structured information can improve the detection performance of the model.

4.3 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[14, 30, 35]) and two deep learning based methods (i.e. TMP and DR-GCN[48]) 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., [31, 38] 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 twenty tools with the highest numbers of *stars* from the search results. Next, we selected the nine tools and two methods based on the following criteria:

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

4.3.1 *Comparison with traditional tools/method.*

Result. Table 5 and Table 6 show the experimental results of the state-of-the-art vulnerability detection tools and methods, where N/A represents that an 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.

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 unpleasant user experience in actual applications.

Table 2: Performance evaluation of the models on *Re-entrancy, Timestamp Dependency* and *Tx.origin*.

Methods	Re-entrancy				Timestamp Dependency				Tx.origin			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
RNN	74.70	81.71	71.28	76.14	77.29	65.83	86.63	74.28	77.19	82.50	74.95	78.43
GRU	83.74	87.96	81.93	84.51	81.67	68.75	85.66	78.89	81.56	81.04	76.53	83.15
LSTM	84.33	86.59	82.56	84.52	82.50	68.33	85.37	79.61	82.50	83.13	76.80	84.18
BLSTM	86.75	84.94	88.32	86.38	87.50	87.92	87.50	87.53	89.84	88.13	85.32	89.68
BLSTM+ATT	**90.84**	**88.47**	**90.11**	**90.60**	**89.17**	**89.37**	**89.20**	**88.29**	**91.56**	**90.00**	**88.24**	**91.43**

Table 3: Performance evaluation of the models on *Integer Overflow and Underflow* and *Unprotected Selfdestruct Instruction*.

Methods	Integer Overflow and Underflow				Unprotected Selfdestruc Instruction			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
RNN	79.47	75.66	72.15	73.86	75.69	71.83	77.51	74.56
GRU	80.95	83.87	78.79	81.25	76.19	78.13	75.76	76.92
LSTM	82.64	83.75	76.41	79.91	77.01	73.35	81.13	77.04
BLSTM	86.45	84.36	85.27	84.81	84.37	82.54	85.16	83.83
BLSTM+ATT	**91.67**	**88.89**	**94.12**	**91.43**	**90.48**	**93.75**	**88.24**	**90.91**

Table 4: 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(%)
Re-entrancy	88.36	83.33	87.22	89.74	85.42	81.78	84.78	84.80	**90.84**	**88.47**	**90.11**	**90.60**
Timestamp	85.83	80.67	79.45	87.22	78.33	83.33	75.76	79.37	**89.17**	**89.37**	**89.20**	**89.20**
Tx.origin	86.88	89.00	80.19	88.24	82.19	83.75	84.76	82.77	**91.56**	**90.00**	**88.24**	**91.43**
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**	**90.91**

Table 5: Performance comparison with the existing tools/methods on *Re-entrancy, Timestamp Dependency* and *Tx.origin*.

Methods	Re-entrancy				Timestamp Dependency				Tx.origin			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
SmartCheck	73.50	70.73	55.44	37.42	70.91	55.89	50.00	61.75	54.38	50.00	52.66	54.01
Conkas	55.42	24.39	37.87	41.90	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Honeybadger	51.40	50.02	60.10	51.92	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Mythril	55.10	44.63	36.22	45.38	54.47	51.79	42.22	43.31	46.17	32.54	22.54	22.69
Osiris	43.72	32.44	40.60	40.96	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Oyente	63.39	54.88	42.06	42.90	64.81	58.04	34.62	53.04	N/A	N/A	N/A	N/A
Securify	76.23	65.37	30.17	44.59	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Slither	78.69	60.49	58.04	45.83	79.89	70.71	62.34	72.18	70.27	61.54	60.55	50.08
Maian	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DR-GCN	80.12	79.89	72.36	75.94	77.83	75.59	70.34	72.87	N/A	N/A	N/A	N/A
TMP	84.82	81.65	79.31	80.46	82.75	83.82	73.24	78.17	N/A	N/A	N/A	N/A
DeepFusion	**90.84**	**88.47**	**90.11**	**90.60**	**89.17**	**89.37**	**89.20**	**89.20**	**91.56**	**90.00**	**88.24**	**91.43**

4.3.2 *Comparison with Deep Learning Methods.*

Here we conduct an in-depth analysis on the deep learning based methods. *DR-GCN* and *TMP* [48] are based on graph neural networks to detect *re-entrancy* and *timestamp dependence* vulnerabilities. They construct contract graphs by 1) extracting control flow and data flow information from smart contract code, 2) extracting representative key function calls or variables as nodes and call relationships between functions as edges, and 3) normalising the proposed 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.

Result. From Table 5 and Tabel 6, we can see that, although the deep learning methods outperform the existing detection tools, the

Table 6: Performance comparison of with the existing tools/methods on *Integer Overflow and Underflow* and *Unprotected Selfdestruct Instruction*.

Methods	Integer Overflow and Underflow				Unprotected Selfdestruct Instruction			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
SmartCheck	68.37	37.17	65.62	47.46	N/A	N/A	N/A	N/A
Conkas	67.69	80.53	55.49	65.70	N/A	N/A	N/A	N/A
Honeybadger	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Mythril	62.89	11.50	61.90	19.40	62.50	12.12	80.00	21.05
Osiris	46.85	12.39	20.90	15.56	N/A	N/A	N/A	N/A
Oyente	51.36	85.84	43.30	57.57	N/A	N/A	N/A	N/A
Securify	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Slither	75.17	47.79	79.41	59.67	65.43	42.42	60.87	50
Maian	N/A	N/A	N/A	N/A	56.63	50.78	44.36	47.35
DR-GCN	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
TMP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DeepFusion	**91.67**	**88.89**	**94.12**	**91.43**	**90.48**	**93.75**	**88.24**	**90.91**

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 coverage of these methods is also not as broad as our approach.

Answer to RQ3: *DeepFusion* performs better than the existing methods and detection tools in terms of Accuracy, Recall, Precision and F1.

4.4 Efficiency

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

Result. Table 7 shows the time used by each tool/method for different vulnerability types, where N/A represents that an analysis tool/method is not designed to detect a certain type of vulnerability. It is not difficult to see 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 pre-processing 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 six tools are about the same level.

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

- The reason why the detection speed of *Slither* and *Smarkcheck* tools is faster is because these two tools are static detection tools so that there is no need to compile and execute smart contracts.
- 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 *TMP* and *DR-GCN* is that *DeepFusion* is 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. However, the time differences are relatively minor and the overall performance of *DeepFusion* is still far better than that of the other two 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 good as the detection method of single representation information, and the gap is not very large, the efficiency is significantly better than that of traditional detection tools.

4.5 Threats to Validity

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

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

Table 7: 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

- *Restricted extensibility*. So far we can employ *DeepFusion* to detect 5 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 3.2). 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.
- *Limited types of detection vulnerabilities*. *DeepFusion* is only able to detect 5 types of vulnerabilities. With the increasing number of Ethereum smart contracts and the change of deployment environment, there emerge more than 5 kinds of smart contract vulnerabilities on Ethereum, according to the study[47]. However, most vulnerability types lack a fair and open dataset for performing the research in this area. Therefore, Therefore, we constructed a dataset containing the five types of vulnerabilities. However, when selecting the vulnerability types, we fully considered their universality and extent of damage. We will expand the scope of vulnerability detection to other types in future research work.

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 of these tools may also affect the validity of *DeepFusion*. Nevertheless, *solc*, *Slither* and *ANTLR* are currently widely used open-source tools, which are regularly maintained by dedicated organizations and developers. This reduces the impact of this threat.

5 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.

Traditional methods. Traditional smart contract vulnerability detection tools are further divided into three categories: symbolic execution, abstract interpretation and fuzzing. Trail of Bits[15] proposes *Slither* based on symbolic execution, which preserves the semantic information of vulnerabilities for error detection by converting Solidity code into an intermediate representation called SlithIR, whose open source github user manual claims to detect more than eighty smart contract vulnerabilities. Tsankov et al.[37] propose *Securify* based on abstract interpretation, compared with the vulnerability standards of SWC and *Slither*, it defines contracts and violation patterns by analyzing the dependency graph of the program, so as to extract the semantic information of vulnerabilities. Tikhomirov et al.[35] propose *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.[27] propose *Oyente*, which constructs a contract control flow graph at the bytecode level. Torres et al.[36] propose *Osiris*, which generates basic blocks of contracts containing integer overflow errors by analyzing CFGs constructed at the bytecode level. Consensys[28] proposed *Mythril*, which simulates contract invocations through multiple symbolic executions for vulnerability detection. 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.

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[49] shows that it has higher accuracy and completeness than traditional detection. Zhuang et al.[48] 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 re-entrancy vulnerability, timestamp dependence vulnerability, and infinite loop vulnerability. However, 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.

6 CONCLUSION

In this paper, 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.

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*. Springer, 164–186.

[2] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

[3] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.

[4] Jiachi Chen, Xin Xia, David Lo, and John Grundy. 2021. Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2021), 1–37.

[5] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering* (2020).

[6] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* (2021).

[7] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 1–44.

[8] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2020. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2020), 1433–1448.

[9] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 442–446.

[10] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th international conference on software engineering: New ideas and emerging technologies results (ICSE-NIER)*. IEEE, 81–84.

[11] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).

[12] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International conference on financial cryptography and data security*. Springer, 79–94.

[13] Wesley Dingman, Aviel Cohen, Nick Ferrara, Adam Lynch, Patrick Jasinski, Paul E Black, and Lin Deng. 2019. Classification of smart contract bugs using the nist bugs framework. In *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 116–123.

[14] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[15] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

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

[17] Cuiyun Gao, Jichuan Zeng, Xin Xia, David Lo, Michael R Lyu, and Irwin King. 2019. Automating app review response generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 163–175.

[18] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering* (2020).

[19] Asem Ghaleb and Karthik Pattabiraman. 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*. 415–427.

[20] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018).

[21] Tharaka Hewa, Mika Ylianttila, and Madhusanka Liyanage. 2021. Survey on blockchain based smart contracts: Applications, opportunities and challenges. *Journal of Network and Computer Applications* 177 (2021), 102857.

[22] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[23] Hai Jin, Zeli Wang, Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou. 2021. Aroc: An Automatic Repair Framework for On-chain Smart Contracts. *IEEE Transactions on Software Engineering* (2021).

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[25] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[26] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[28] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9 (2018), 54.

[29] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[30] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.

[31] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. *arXiv preprint arXiv:1809.02702* (2018).

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

[33] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *IEEE Access* 8 (2020), 24416–24427.

[34] Saurabh Singh, ASM Sanwar Hosen, and Byungun Yoon. 2021. Blockchain security attacks, challenges, and solutions for the future distributed iot network. *IEEE Access* 9 (2021), 13938–13959.

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

[36] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676.

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

[38] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: a practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1422.

[39] Ben Wang, Hanting Chu, Pengcheng Zhang, and Hai Dong. 2021. Smart Contract Vulnerability Detection Using Code Representation Fusion. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 564–565.

[40] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2020. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2020), 1133–1144.

[41] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[42] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 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*. 378–389.

[43] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1029–1040.

[44] Xingxin Yu, Haoyue Zhao, Botao Hou, Zonghao Ying, and Bin Wu. 2021. DeeSCVHunter: A Deep Learning-Based Framework for Smart Contract Vulnerability Detection. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[45] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[46] Meng Zhang, Pengcheng Zhang, Xiapu Luo, and Feng Xiao. 2020. Source Code Obfuscation for Smart Contracts. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 513–514.

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

[48] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network.. In *IJCAI*. 3283–3290.

[49] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing* (2019).

[50] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.