Smart Contract Vulnerability Detection based on Multi-scale Encoders

Junjun Guo * , Long Lu , Jingkui Li

Posted Date: 3 January 2024

doi: 10.20944/preprints202312.2325.v1

Keywords: Smart contract; deep learning; Multi-scale; vulnerability detection

Article

Smart Contract Vulnerability Detection based on Multi-Scale Encoders

Junjun Guo *, Long Lu and Jingkui Li

School of Computer Science and Engineering, Xi'an Technological University, Xi'an, Shaanxi 710021, China
* Correspondence: guojunjun@xatu.edu.cn

Abstract: Vulnerabilities in smart contracts may trigger serious security events, and the detection of smart contract vulnerabilities has become a significant problem. In this paper, by using the multi-scale cascade encoder architecture as the backbone, we propose a novel Multi-scale Encoder Vulnerability Detection (MEVD) approach to detect well-known high-risk vulnerabilities in smart contracts. Firstly, we use the gating mechanism to design a unique Surface Feature Encoder (SFE) to enrich the semantic information of code features. Then, by combining a Base Transformer Encoder (BTE) and a Detail CNN Encoder (DCE), we introduce a dual-branch encoder to capture the global structure and local detail features of the smart contract code, respectively. Finally, to focus the model's attention on vulnerability-related characteristics, we employ the Deep Residual Shrinkage Network (DRSN). Experimental results on three types of high-risk vulnerability datasets demonstrate performance compared to state-of-the-art methods, and our method achieves an average detection accuracy of 90%.

Keywords: smart contract; deep learning; multi-scale; vulnerability detection

1. Introduction

Blockchain is a distributed ledger that records the finance transactions that happen within decentralized networks [1]. With the emergence of Ethereum smart contracts, blockchain has entered the era of programmable finance [2]. Smart contracts were initially proposed for the dissemination, validation, and enforcement of contracts in an informational manner [3]. Compared to traditional contracts, smart contracts enabled users to codify their agreements and trust relations by providing automated transactions without the supervision of a central authority [4].

With the widespread application of smart contracts in finance, industry, commerce and other fields, the security of smart contracts is becoming increasingly important [5]. If there are vulnerabilities in the smart contract, attackers can exploit these smart contract vulnerabilities to maliciously penetrate blockchain networks and steal tens of millions of dollars cryptocurrency. This not only has implications for the reliability of the blockchain system, but also serious for the interests of smart contract holders. There is a typical event that attacker exploited a reentrancy vulnerability to successfully abscond with millions of ether called the DAO attack [6]. This event seriously undermined the credibility of the blockchain system. Events based on smart contract vulnerability attacks are still occurring continuously [7], Undoubtedly, these attacks highlight the urgent need for smart contract vulnerability detection. By detecting potential vulnerabilities in a smart contract, the security and stability of the blockchain system can be effectively improved [8]. Therefore, how to design efficient vulnerability detection models is an important problem and challenge that needs to be solved.

Traditional methods to detect vulnerabilities in smart contracts suffer from the problem of path explosion, which leads to a large increase in the execution time and memory overhead. Moreover, these methods cannot effectively parse complex data structures of program, which may result in higher false positive and false negative rates. In recent years, researchers have used deep learning techniques to improve the accuracy of detecting vulnerabilities in smart contracts [9]. By segmenting

 doi:10.20944/preprints202312.2325.v1

the source code of smart contracts slice into code snippets and using the BLSTM model, Qian et al. designed ReChecker to detect reentrancy vulnerabilities [10]. However, the defined slice criteria are not comprehensive, and the generated code snippets may implicitly ignore critical segments. In addition, for longer code snippets, BLSTM may not efficiently capture global semantic information [11], all this will lead to false positives. Based on CNN and BIGRU network models, Zhang et al. devised a CBGRU hybrid deep learning model to detect vulnerabilities [12]. However, due to CNN's inability to fully extract local syntax and semantic detail features and BIGRU's limited perception of global structural features, the detection results can lead to false negatives. To detect reentrancy vulnerabilities, Wu et al. proposed a Peculiar approach for extracting critical data flow graphs from smart contract code [13]. However, this method cannot detect other types of vulnerabilities and has a lack of generality.

To overcome the above shortcomings, we proposed a novel Multi-Scale Encoder Vulnerability Detection (MEVD) approach. Firstly, due to the original feature contain excessive semantic information that is irrelevant to vulnerability features [14], which affects the correlation of the global semantic information and the coherence of the contextual structure. To address these problems, we have designed a Surface Feature Encoder (SFE) based on gating mechanism. The unique mechanism can control the flow of critical semantic information, suppress noise data, and enhance the global semantic information of the features. Second, by combing a Base Transformer Encoder (BTE) based on Transformer blocks and Detail CNN Encoder (DCE), we devise a dual-branch structured encoder to comprehensively explore vulnerability patterns from global structure and local detail features, where the BTE is used to extract long-range dependencies and global structural features in the code, the DCE focuses more on the analysis and exploration of the syntax and semantics within the code to extract local detailed features. Third, to improve the quality of the code vector representation, we introduce Deep Residual Shrinkage Networks (DRSN) to resist the noise generated by vulnerability unrelated variables. By using deep learning to automatically determine the threshold, the model can focus more on the vulnerability-related characteristics. Due to the large amount of irrelevant information contained in the contract code [15], in data preprocessing phase, we use Vulnerability Syntax-driven Slice (VSS) to eliminate redundant information. VSS provides a more comprehensive set of slice criteria for identifying different vulnerability characteristics, which can maintain data dependency and control relevance between code statements. We conducted extensive experiments on three different types of vulnerability datasets: Reentrancy, Timestamp Dependency and Infinite Loop vulnerabilities. The experimental results show that our approach outperforms state-of-the-art methods in terms of detection performance. **The major contributions of this paper are as follows:**

1. We present a Vulnerability Syntax-driven Slice (VSS) method, which simplifies the contract code by removing statements unrelated to vulnerability characteristics and preserving the data and control dependencies in statements.
2. We propose a novel Multi-scale Encoder Vulnerability Detection (MEVD) approach to detect vulnerabilities in smart contracts, global structure and local detail features are captured by multi-scale encoders, respectively, to compensate for the lack of feature extraction capability of a single model.
3. We have compared MEVD with state-of-the-art vulnerability detection methods. Experimental results show that the proposed MEVD outperforms existing methods for detecting Reentrancy, Timestamp Dependency, and Infinite Loop vulnerabilities.

This paper is organized as follows. First, in Section 2, we discuss the background knowledge of smart contract vulnerabilities. Next, in Section 3, we introduce related work. In Section 4, we provide a detailed explanation of our proposed method. Section 5 presents our experimental results. Finally, we conclude the paper by discussing future directions.

2. Background

In this section, we introduce background about the smart contract and common vulnerabilities in the smart contract.

 doi:10.20944/preprints202312.2325.v1

2.1. Smart Contract Basics

Smart Contracts: Smart contracts are typically written in the Solidity programming language. When smart contracts are deployed on the blockchain, transactions can be executed automatically and irreversibly. Smart contracts ensure the security and reliability of transactions without the need for third party verification and execution. This effectively reduces transaction costs and risks.

Fallback Function: In a smart contract, when an unmatched function is called by an external user, the fallback function is automatically executed to handle unknown transactions or calls [16]. It can be used to automatically receive transfers of Ether or other digital assets when performing funds transfers operation, enabling smart contracts to accept funds and execute related business logic.

2.2. Smart Contract Vulnerabilities

Vulnerabilities in smart contracts exist at three levels: Solidity language, EVM virtual machine, and blockchain [16]. For example, at the Solidity language level, vulnerabilities types include Reentrancy, Integer Overflow, and Underflow; at the EVM virtual machine level, vulnerabilities types include Short Address Attack and Storage Overlap Attack, at blockchain level, vulnerabilities types include Timestamp Dependency, Transaction Order Dependency, and so on. Our work is mainly focused on Reentrancy, Timestamp Dependency and Infinite Loop vulnerabilities. By checking 40,932 Ethereum smart contracts, Liu et al. discover out of 307,396 functions, where approximately 5,013 functions may lead to reentrancy vulnerabilities, and about 4,833 functions may result in timestamp dependency vulnerabilities [17]. In addition, around 56,800 functions contain for or while loop statements, which can lead to infinite loop vulnerabilities. These scenes have led us to investigate on these three types of vulnerabilities.

Reentrancy: The reentrancy vulnerability is a commonly exploited vulnerability in blockchain networks, with the well-known the DAO attack incident being an attack that exploited this vulnerability, attackers can make a program execute malicious code designed by repeatedly in a transaction until the victim's account balance is 0 or gas is exhausted [9], thus causing huge financial losses. This allows attackers to illegally acquire a significant amount of funds within the contract.

Timestamp Dependency: As the system determines the timestamp during the mining process, it allows for a deviation of 900 seconds so that the miners can control the timestamp to a certain extent [18]. However, malicious miners can set arbitrary values for timestamps within a short time frame (< 900 seconds), they can gain foreknowledge of the timestamp of the next block, and thus manipulate the smart contract to gain illegal profits.

Infinite Loop: An infinite loop vulnerability is where the code within a contract function contains an iterative or looping structure (such as a for loop, while loop, etc.) but lacks a termination condition or the termination condition cannot be met, resulting in the loop not being able to terminate normally [17]. This can trap a smart contract in an infinite loop state, consuming the contract's computational resources and potentially causing the contract to fail to execute.

The damage caused by these three vulnerabilities is very serious, and the logic of the contract code becomes more complex due to the numerous call relationships between contracts, they are not easily discovered by developers. Therefore, detection of these three vulnerabilities is essential.

3. Related Work

In this section, we will discuss the current smart contract vulnerability detection methods, including traditional detection methods and deep learning-based methods. Then, we will introduce some deep learning models.

Traditional detection Methods: Traditional methods for detecting vulnerabilities in smart contracts primarily include methods such as symbolic execution, formal verification [19], and fuzzing [9]. Oyente leverages the static symbolic execution technique to detect potential vulnerabilities in smart contracts [20]. Mythril combines static symbolic execution, taint analysis and control flow checking to further improve the accuracy of vulnerability detection [21]. Slither can detect

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

vulnerabilities by transforming contract code into an intermediate representation [22]. ContractFuzzer is the first vulnerability detection framework to apply fuzz testing techniques to smart contracts [18].

Deep Learning detection Methods: In recent years, deep learning technology has achieved remarkable success in various fields. Currently, there are a number of deep learning methods for vulnerability detection. ReChecker [10] employs code slicing of smart contract source code and uses a BLSTM-ATT sequence model to detect reentrancy vulnerabilities. To fully capture key statements associated with the vulnerability, Yu et al. had made improvements to the slicing technique by introducing the concept of Vulnerability Candidate Slices (VCS), which feed into various temporal neural network models for vulnerability detection [23]. Cai et al. constructed a novel contract graph, incorporating Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Program Dependency Graphs (PDGs) and employed graph neural network models for vulnerability detection [11]. Liu et al. introduced a new method for representing code graphs, AME classifies nodes in the graph into core and normal nodes, then extracts deep graph features and fuses global graph features with local expert patterns to improve accuracy [24]. VulnSense combines three types of features from smart contracts and uses a multimodal learning approach for a comprehensive vulnerability detection method [25].

Deep Learning Models: In recent years, many outstanding network models have emerged, such as Long Short-Term Memory (LSTM [26]), Graph Convolutional Networks (GCN [27]), and Convolutional Neural Networks (CNN). These models have achieved significant success in their respective domains. Above In addition to these basic models, more complex model structures have emerged in recent years, such as the Transformer model [28]. The core components of the Transformer model are encoders and decoders that use self-attention mechanisms to establish global correlations within input sequences, thereby better capturing contextual structural information. The Transformer has made significant breakthroughs in natural language processing, particularly machine translation. By refining the Transformer's encoder and decoder modules, Restormer better captures interactions between distant pixels [29]. RepVGG introduces a multi-branch structure to extract richer features [30]. DRSN is a network structure to improve the feature learning capability for highly noisy vibration signal analysis, enabling more accurate fault diagnosis [31]. CDDFuse designs a novel autoencoder to address the problem of multimodal image fusion [32].

4. Method

In this section, we introduce a novel method for slicing smart contract code and also present the design of the MEVD framework.

4.1. Overview

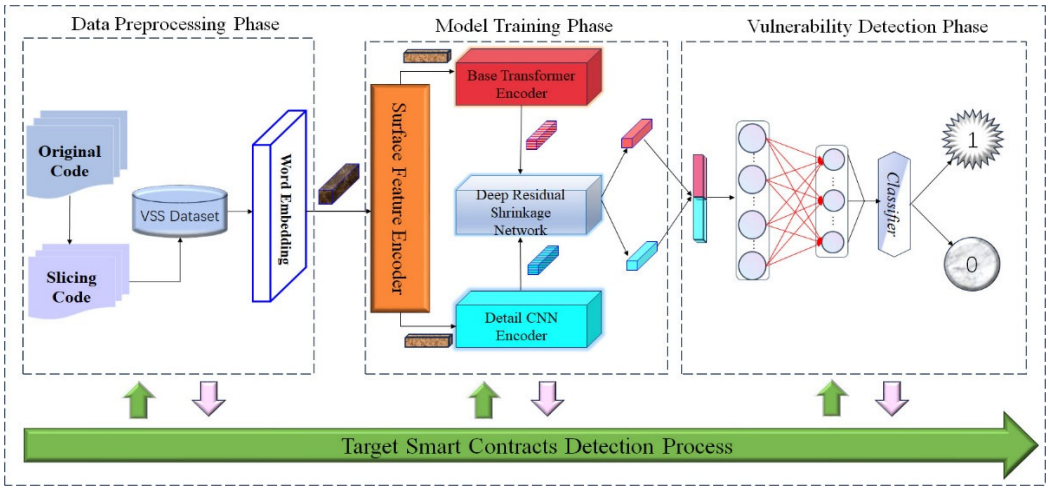

Figure 1. Multi-Scale Encoder Vulnerability Detection (MEVD) framework and vulnerability detection process.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

As shown in Figure 1, the MEVD proposed in this paper consists of three phases: data preprocessing phase, model training phase, vulnerability detection phase. Firstly, in the data preprocessing phase, we remove redundant information from the smart contract code, such as blank lines, comments and non-ASCII characters. Then, to simplify the contract, the source code is sliced into Vulnerability Syntax-Driven Slice (VSS), and uniform naming rules for normalization. Next, we use the Word2Vec word embedding technique [33] to convert code slices into vector forms suitable for input to the model in the model training phase. The model consists of multi-scale encoders that are used to enrich the semantic information of the feature vectors and capture global structural and local detail features that are used to improve the performance of vulnerability detection. Finally, in the vulnerability detection phase, to obtain a vector containing global structure and local detail features, the two feature vectors processed by the multiple encoders are fed into a fusion layer. These feature vectors are then fed into a fully connected layer and a softmax activation function is used to output vulnerability detection probabilities.

4.2. Data Preprocessing

4.2.1. Code Slice Criteria

It has been discovered that there are a number of code statements in the contract source code that are unrelated to vulnerabilities [34]. These irrelevant code statements can have an impact on vulnerability detection performance. It is therefore important to extract key code statements accurately. However, existing methods for the definition of slice criteria have certain limitations when dealing with smart contract code, which can result in the loss of implicit information about vulnerabilities. Therefore, as shown in Table 1, we propose an entirely new set of slice criteria that cover three types of vulnerabilities.

Table 1. The different vulnerability slice criteria.

Vulnerability	Slice Criteria
Reentrancy	*fallback ()* *call.value()* *a function involving call.value* *variable: correspond to user balance*
Timestamp Dependency	*block.timestamp* *block.number* *now*
Infinite Loop	*for* *while loop* *self-call function*

Taking the reentrancy vulnerability as an example, we have defined several slice criteria. Firstly, code statements that include *call.value()* and *fallback()* are considered to be critical code statements that are susceptible to reentrancy vulnerabilities, because they allow smart contracts to go through state transitions as they interact with other contracts. This state transition could result in the contract continuing to execute after an external call, which may result in potential vulnerabilities. In addition, funds transfer functions that include *call.value* calls also require special attention. These functions are widely used in the transfer and invocation processes of smart contracts and serve as conditions for launching reentrancy attacks. Then, we have also focused on variables related to user balances, as these variables are equally important in fund transfers. Similarly, we have also defined slice criteria for the other two types of vulnerabilities. This novel set of slice criteria enables more comprehensive vulnerability-related information to be captured within the contract code, avoiding the omission of critical implicit details.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

4.2.2. Code Slice Generation

To address the impact of redundant information in smart contract source code on the accuracy of vulnerability detection, we have proposed a novel Vulnerability Syntax-Driven Slice (VSS) method, as shown in Figure 2.

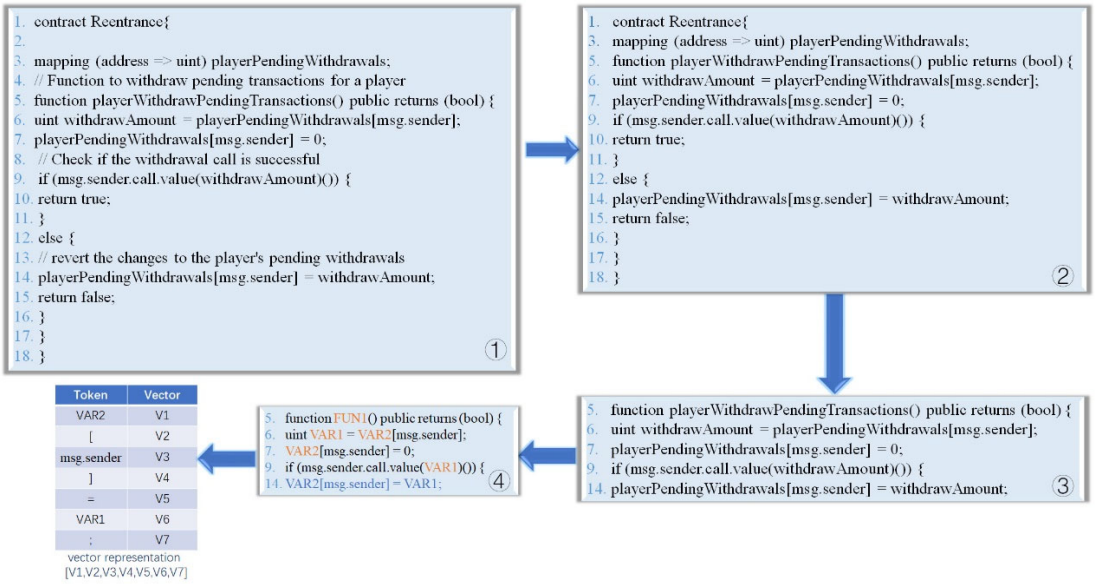

Figure 2. The process of generating code slices from the source code and transforming them into vectors.

Taking a smart contract containing *call.value()* as an example, firstly, we remove comments, non-ASCII characters, and blank lines from the smart contract. Removing this irrelevant information does not affect the results. Next, based on the vulnerability type associated with the code, we identify the key statements of the code according to the slice criteria in Table 1. Then, the code statements relevant to the key statements are extracted and organized into code snippets by analysis of the control and data dependencies between statements and variables. We then map user-defined variables to symbolic names (e.g., "VAR1", "VAR2") and user-defined functions to symbolic names (e.g., "FUN1", "FUN2"), this is the final generated VSS. Finally, to convert code snippets into a format that can be accepted by neural networks, we use Word2Vec technology to convert code snippets into vector representations.

4.3. MEVD Model

In this section, we will introduce the model structure of MEVD. The MEVD model consists of three modules: the Surface Feature Encoder (SFE), the Dual Branch Transformer-CNN Encoder and the Deep Residual Shrinkage Network (DRSN).

4.3.1. Surface Feature Encoder

We used the word2vec algorithm, which can convert tokens to vectors, but by converting code directly to vectors as input, semantic features within the code may be ignored. To solve this problem, we designed SFE, which can enhance the relevance of global semantics within the features and suppress the influence of irrelevant information, thus preserving richer syntactic semantic information. The detailed design of the SFE module is shown in Figure 3.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

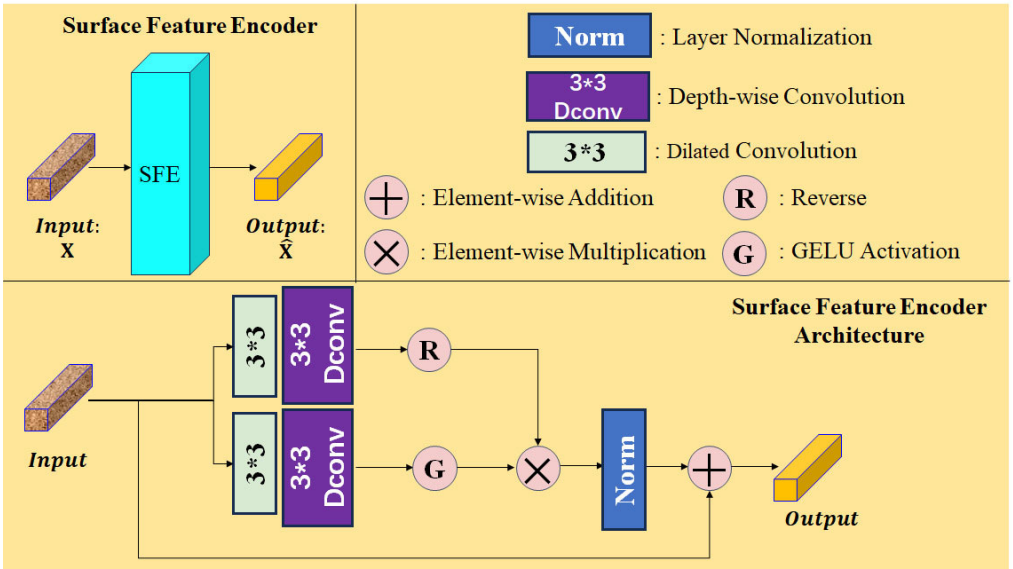

Figure 3. The architectural framework of SFE.

SFE leverages three design strategies to enhance semantic features: a gating mechanism, dilated convolutions, and depth-wise convolutions. Gating mechanisms are typically used to control the flow of information by enhancing attention to critical information. Specifically, the gating mechanism is formulated as the element-wise product of two parallel paths of linear transformation layers. One path is activated by the non-linear activation function GELU, while the other path applies the reverse operation to the vector. The outputs of these two paths are then subjected to element-wise multiplication, allowing information from one path to modulate the strength of the other, thereby enhancing focus on important semantic details and reducing the impact of irrelevant variables, On the other hand, the SFE employs dilated convolutions and depth-wise convolutions. Dilated convolutions extend the receptive field to cover a wider range of input information, helping to capture different scales and structural features within the code. Meanwhile, depth-wise convolutions focus on the local context and detail information in the input data. They efficiently capture local interdependencies between different channels in the input data, helping the SFE to enrich the semantic information within the features. Given the input vector (feature matrix) $\mathbf{X} \in \mathbb{R}^{L\times D}$, where L represents length of the token sequence, and D represents the dimension of the token embedding. SFE is the equation:

$$\text{Gating}(\mathbf{X}) = \phi(W_{de}^{1}W_{di}^{1}(\mathbf{X})) \odot Reverse(W_{de}^{2}W_{di}^{2}(\mathbf{X})) \quad (1)$$

$$\hat{\mathbf{X}} = LN\left(\text{Gating}\ (\mathbf{X})\right) + \mathbf{X} \quad (2)$$

Where $w_{di}^{(\cdot)}$ is the 3×3 dilated convolution and $W_{de}^{(\cdot)}$ is the 3×3 depth-wise convolution. where $\odot$ denotes element-wise multiplication, ϕ represents the GELU non-linearity, and LN is the layer normalization. *Reverse* denotes the reversal of rows in a feature matrix, e.g. swapping the first row with the last. In summary, the design of the SFE allows each level to be concerned with details that are complementary to other levels, focusing on enriching features with code semantic context information.

4.3.2. Dual-Branch Encoder

The Dual Branch Encoder consists of two parallel encoders: the Base Transformer Encoder (BTE) and the Detail CNN Encoder (DCE). These two modules are used to further extract vector features after the SFE. The BTE is used to effectively capture global structural information, allowing it to capture dependencies between code statements and contextual logical structural features. The DCE, on the other hand, focuses on exposing crucial latent features within the code, while preserving more syntactic and semantic details. This dual-branch design strategy allows the model to simultaneously consider global structure and fine-grained features, leading to a more comprehensive understanding

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

of smart contract code and significantly improving the accuracy and performance of vulnerability detection.

a. Base Transformer Encoder (BTE)

The BTE consists of multiple self-attention layers, with each self-attention layer containing multi-head self-attention and feedforward networks. It uses the multi-head self-attention feature of the Transformer encoder to capture contextual information and long-range dependencies within code sequences, thereby extracting richer global structural features. The multi-head self-attention mechanism serves as the core of BTE, allowing it to establish correlations between different positions in the input sequence and effectively capture contextual information. The structure of the BTE is shown in Figure 4.

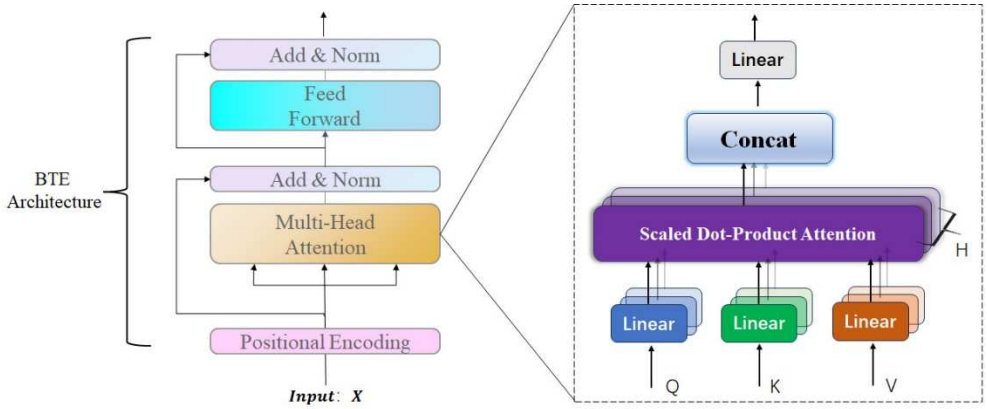

Figure 4. The architectural framework of BTE.

BTE achieves parallel computation of multiple attentional heads, which allows the model to simultaneously focus on information from different positions within the input sequence. It then combines the outputs of the different heads using weighted fusion to obtain a more comprehensive feature representation. The calculation process of the BTE encoder is as follows:

$$X_{pos} = \text{Positional\_Encoding}\ (X) \quad (3)$$

$$Q = \text{Linear}(X_{pos}) = X_{pos} * W_Q \quad (4)$$

$$K = \text{Linear}(X_{pos}) = X_{pos} * W_K \quad (5)$$

$$V = \text{Linear}(X_{pos}) = X_{pos} * W_V \quad (6)$$

$$X_{\text{attention}} = \text{Self\_Attention}(Q, K, V) \quad (7)$$

X_{pos} is obtained using positional encoding operations, Q represents the query vector, K represents the key vector, and V represents the value vector. Each vector undergoes a linear transformation, with W_Q, W_K, and W_V representing the weight matrices for these transformations. Finally, $X_{\text{attention}}$ is obtained by the Equation (7).

$$X_{\text{attention1}} = X_{pos} + X_{\text{attention}} \quad (8)$$

$$X_{\text{attention2}} = \text{LayerNorm}\ (X_{\text{attention1}}) \quad (9)$$

$$X_{\text{output}} = Activate(Linear(Linear(X_{\text{attention2}}))) \quad (10)$$

$$X_{\text{output1}} = X_{\text{attention2}} + X_{\text{output}} \quad (11)$$

$$X_{\text{output2}} = \text{LayerNorm}\ (X_{\text{output1}}) \quad (12)$$

Then, the $X_{\text{attention2}}$ is obtained by residual connection and Layer Normalization, Finally, $X_{\text{attention2}}$ processed through a feedforward neural network and residual connection to obtain X_{output2} feature vectors with global structure and semantic information. This design enables the model to better understand the overall structure and global semantic relationships of smart contract code and effectively handle complex smart contracts.

b. **Detail CNN Encoder (DCE)**

BTE focuses primarily on capturing global semantic information and structural features, while DCE complements BTE by focusing on extracting more intricate and complex features. DCE has the ability to delve deep into the latent syntactic, semantic and local features of the code, further enriching

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

the feature representation of the code vectors. DCE consists of a multi-branch convolutional structure as shown in Figure 5, consisting of three branches: a 3×3 dilated convolution layer with Batch Normalization (BN), a 1×1 dilated convolution layer with BN, and an independent BN layer. Dilated convolution allows different dilation rates to be set, providing a larger receptive field without increasing the number of parameters. This helps the model to better understand the structure and relationships within the code, thus capturing more intricate details. In addition, the different kernel sizes in these convolutions allow the DCE to capture information at different scales. Finally, by fusing outputs from different branches, it integrates information from different scales and levels of detail. This helps to improve the model's perception of fine-grained details and to combine features from different hierarchical levels for a better understanding of the detailed features of the code.

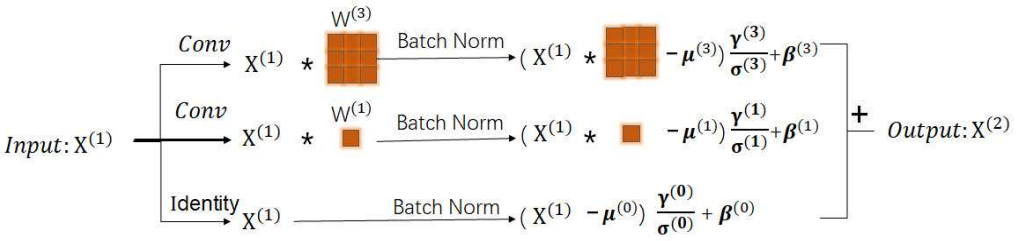

Figure 5. The architectural framework of DCE.

In this research, we utilize symbols to represent key concepts. $W^{(3)} \in \mathbb{R}^{3\times3}$ represents the convolution kernel for a 3×3 dilated convolution layer, while $W^{(1)} \in \mathbb{R}^{1\times1}$ represents the convolution kernel for a 1×1 dilated convolution layer. Additionally, we use $\mu^{(\cdot)}$, $\sigma^{(\cdot)}$, $\gamma^{(\cdot)}$, and $\beta^{(\cdot)}$ to respectively denote the accumulated mean, standard deviation, learned scaling factor, and bias for each branch in the BN layer. We have also defined symbolic representations for the input and output. $X^{(1)}$ represents the input, and $X^{(2)}$ represents the output. The DCE equation is as follows:

$$\begin{aligned} X^{(2)} &= \mathrm{bn}(X^{(1)} * W^{(3)}, \mu^{(3)}, \sigma^{(3)}, \gamma^{(3)}, \beta^{(3)}) \\ &+\mathrm{bn}(X^{(1)} * W^{(1)}, \mu^{(1)}, \sigma^{(1)}, \gamma^{(1)}, \beta^{(1)}) \\ &+\mathrm{bn}(X^{(1)}, \mu^{(0)}, \sigma^{(0)}, \gamma^{(0)}, \beta^{(0)}) \end{aligned} \tag{13}$$

In summary, the use of DCE enhances the model's ability to perceive and capture these intricate and complex features, thereby helping the model to better distinguish between vulnerable code and normal code.

4.3.3. Deep Residual Shrinkage Network

The DRSN module integrates soft thresholding and deep learning, introducing a method for adaptive thresholding [31]. Through DRSN, the model can effectively reduce irrelevant information during feature learning. The calculation of the soft threshold is as follows:

$$y = \begin{cases} x - \tau & x > \tau \\ 0 & -\tau \le x \le \tau \\ x + \tau & x < -\tau \end{cases} \tag{14}$$

The soft thresholding calculation is shown in Figure 6b, where x represents the input features, y represents the output features, and τ is the threshold, which is a positive parameter. As can be seen from the Equation (14), the essence of soft thresholding is to discard features with smaller absolute values and shrink features with larger absolute values, thereby reducing information unrelated to the current task.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

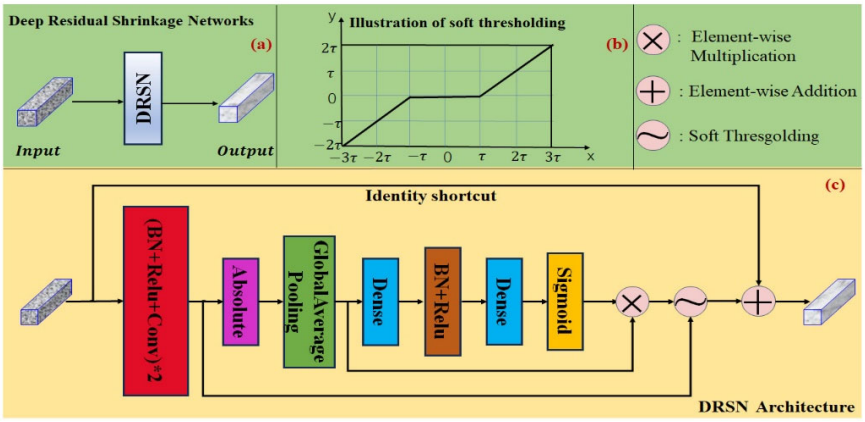

Figure 6. The architectural framework of DRSN and Soft Thresholding Principle.

The structure of DRSN is shown in Figure 6c. First, the input vector passes two convolution layers, each accompanied by normalization layers and Relu activation functions. If the output of the convolution layer is a matrix is denoted as $X \in R^{T\times K}$, where T is the number of token sets, and K is the number of convolution kernels. The resulting matrix undergoes an absolute value operation, followed by global average pooling to generate a one-dimensional vector, denoted as $X_{avg} \in R^{K}$.

$$X_{avg} = \frac{1}{T}\sum_{i=1}^{T} | X_i | \quad (15)$$

Where $| X_i |$denotes the absolute value of the i^{th} row feature in X. Subsequently, X_{avg} is fed through a fully connected layer, followed by batch normalization, a Relu activation function, and a second fully connected layer. Further, a sigmoid activation function is used to scale the threshold parameter for each channel in the range (0, 1), resulting in the scaling parameter $sc \in R^{K}$, and the threshold $\tau \in R^{K}$ can be calculated using Equation (17).

$$sc = \frac{1}{1+e^{-Z}} \quad (16)$$

$$\tau = sc * X_{avg} \quad (17)$$

After processing by DRSN, which eliminates residual redundant information and ensures that the feature vectors retain only crucial information related to the vulnerability characteristics. This helps to improve the performance of the model, enabling it to better distinguish between vulnerable and non-vulnerable samples, thereby increasing the accuracy and effectiveness of vulnerability detection.

4.4. Vulnerability Detection

In the model training phase, we obtain two feature vectors: one from the BTE, denoted as vector B and another from the DCE, denoted as vector D. These two vectors are individually subjected to soft thresholding processing through DRSN, resulting in processed feature vectors B' and D'. To concatenate these two vectors, we flatten these two vectors as 1D vectors and we concatenate them to $B'D'$, which is then fed into a fully connected layer and a Relu activation function for non-linear transformation. Finally, the transformed vector is fed into a softmax layer to determine whether the code snippet contains vulnerabilities. The equations are as follows:

$$B'D' = \text{concat}(Flatten(B'), Flatten(D')) \quad (18)$$

$$\hat{y}_c = softmax(FC(B'D') \quad (19)$$

$\hat{y}_c$ represents the predicted probability, FC represents the fully connected layer, *concat* signifies vector concatenation operation, and $softmax$ denotes the softmax function, used to transform real-valued vectors into probability vectors, indicating the probabilities of each category in a classification problem.

The MEVD model proposed in this paper is a framework that integrates several different functional encoders. In addition, we introduce a novel code-slicing method for data pre-processing. Through an extensive series of experiments, the method proposed in this paper demonstrates outstanding performance in the smart contract vulnerability detection. In the following sections, we

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

present the experimental results to comprehensively substantiate the effectiveness of our proposed approach.

5. Experiments

In this section, we introduced the datasets and configuration parameters used in the experiments. We provided a detailed description of the experimental procedures and presented the experimental results. To evaluate the performance of our proposed method, we answered the following questions:

RQ1: Can the proposed method effectively detect Reentrancy, Infinite Loop, and Timestamp Dependency vulnerabilities? How does its Accuracy, Precision, Recall, and F1-score compare to state-of-the-art vulnerability detection methods?

To answer this question, we compare with state-of-the-art detection tools. Through these comparisons, we were able to assess the performance of our approach within the existing technology.

RQ2: Is the proposed code slicing method helpful for vulnerability detection?

To answer this question, we divided the dataset into two parts: the original dataset and the Vulnerability Syntax-Driven Slice (VSS) dataset. We conducted experiments using these two datasets as model inputs and compared their performance separately.

RQ3: What is the impact of the different components designed in the model on detection performance?

To answer this question，we conducted experiments to compare the performance of models with different combinations of components in terms of metrics such as Accuracy, Precision, Recall, and F1-score.

5.1. Datasets

Our research dataset covers three types of vulnerabilities: Reentrancy vulnerabilities, Timestamp Dependency vulnerabilities, and Infinite Loop vulnerabilities. The primary sources of these datasets include: **(i)** The SmartBugs Wild Dataset consists of 47,398 Solidity language files containing a total of approximately 203,716 contracts with known vulnerabilities [35]. **(ii)** The ESC (Ethereum Smart Contracts) dataset consists of 307,396 smart contract functions extracted from 40,932 smart contracts. **(iii)** In addition, we also use the datasets published by Qian et al. [36].

5.2. Experimental Settings

All experiments were performed on a computer equipped with 32 GB RAM and a GPU at 1080Ti. We implemented our method using the Keras and TensorFlow frameworks. In our experimental, 80% of the contracts form the training set, and the remaining 20% form the test set. The parameters used in the model are detailed in Table 2.

Table 2. Parameters of MEVD model.

Model parameter	Value
Optimizer	Adam
Learning rate	[0.0001, 0.005]
Dropout rate	0.5
Batch size	8
Epochs	50

5.3. Evaluation Metrics

To measure the performance of our approach, we use the following four widely used evaluation metrics:

Accuracy: It represents the proportion of correctly predicted samples out of the total number of samples and measures the overall accuracy of the model's predictions.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \quad (20)$$

Precision: It represents the proportion of true positives among those predicted to be positive by the model and measures the accuracy of the model in predicting positive samples.

$$Precision = \frac{TP}{TP+FP} \quad (21)$$

Recall: It represents the proportion of true positives among the actual positive samples and measures the ability of the model to detect positive samples.

$$Recall = \frac{TP}{TP+FN} \quad (22)$$

F1-Score: This is a composite score that combines Precision and Recall to assess the overall performance of the model.

$$F1 - Score = 2 * \frac{\text{Preccison*Recall}}{\text{Preccison+Recall}} \quad (23)$$

5.4. Results Analysis

Result for Answering RQ1: To evaluate the performance of MEVD in detecting these three types of vulnerabilities, we compared it with deep learning-based approaches (ReChecker [10], DR-GCN [36], TMP [36], CGE [17]). In Table 3, we show the comparison with state-of-the-art deep learning-based methods. These comparisons allow us to assess the performance of our proposed MEVD in vulnerability detection.

Table 3. Comparison with Deep Learning-based Method.

Vulnerability	Method	Acc (%)	Pre (%)	Rec (%)	F1 (%)
Reentrancy	ReChecker	74.65	69.47	76.93	75.62
	DR-GCN	82.58	75.39	80.83	79.57
	TMP	86.28	78.25	84.96	80.28
	CGE	88.32	84.15	86.77	85.64
	MEVD	**92.13**	**89.49**	**90.15**	**89.28**
Timestamp Dependency	ReChecker	72.68	70.86	69.24	73.21
	DR-GCN	80.21	76.54	78.86	79.04
	TMP	84.37	85.21	80.79	83.08
	CGE	87.76	86.15	82.77	84.56
	MEVD	**90.85**	**89.17**	**92.98**	**91.04**
Infinite Loop	ReChecker	68.79	71.63	69.41	71.33
	DR-GCN	76.55	72.25	74.21	78.62
	TMP	80.86	78.05	84.86	81.77
	CGE	85.89	82.62	83.19	83.25
	MEVD	**86.94**	**84.15**	**85.63**	**86.21**

Based on the results presented in Table 3, we can compare the performance of different models in terms of detection. The following conclusions can be drawn from these data: Firstly, the ReChecker model shows relatively poor detection performance compared to other models, particularly in the context of identifying infinite loop vulnerabilities, where its accuracy is only 68.79%. However, when we use the MEVD model for detection, the accuracy improves significantly by 18.15%. It is worth noting that our approach not only outperform in detecting infinite loop vulnerabilities, but also provides satisfactory performance improvements in other metrics. For example, in detecting timestamp dependency vulnerabilities, the accuracy of the MEVD model is 3.09% higher than that of the graph neural network-based CGE. In addition, the MEVD model achieves an accuracy of 92.13% for the detection of reentrancy vulnerabilities, which is well above the performance of other methods.

By comparing the experimental results, we come to a clear conclusion: the MEVD shows outstanding performance in different types of vulnerability detection. The MEVD shows significant performance advantages over state-of-the-art deep learning-based detection approaches.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 January 2024 doi:10.20944/preprints202312.2325.v1

Result for Answering RQ2: To evaluate the effectiveness of our proposed code slicing method, we divided the dataset into two parts: the unprocessed original dataset and the VSS dataset (VSS+Dataset). We then performed experiments on these two datasets separately, as shown in **Table 4**. This approach allows us to evaluate the impact of code slicing on model performance.

Table 4. Performance Comparison of Sliced and Original Data.

Dataset	Acc (%)	Pre (%)	Rec (%)	F1 (%)
Reentrancy	86.32	82.94	85.61	84.49
Timestamp Dependency	85.21	81.54	83.86	86.24
Infinite Loop	83.05	77.63	78.67	81.09
VSS +Reentrancy	92.13	89.49	90.15	89.28
VSS +Timestamp Dependency	90.85	89.17	92.98	91.04
VSS +Infinite Loop	86.94	84.15	85.63	86.21

By analyzing the results in Table 4, we observe that running experiments on the sliced dataset, as opposed to the unprocessed dataset, significantly improves the performance of the models. For example, in the case of reentrancy vulnerability detection, experiments conducted on the sliced dataset yielded an accuracy of 92.13%, which is a significant improvement of 5.81% compared to the unprocessed dataset. At the same time, in the detection of vulnerabilities related to timestamps, the F1-score reaches 91.04%, which represents an improvement of 4.8%. By comparing the experimental results, we arrive at a definitive conclusion: The slice criteria defined by our approach are significant in covering crucial statements within the contract code. Furthermore, by extracting sliced code using control flow and data flow methods, we can expose critical code structure information and deepen the relationships between semantics. This helps models learn vulnerability patterns more effectively.

Result for Answering RQ3: In order to evaluate the effectiveness of different modules in the MEVD, we performed substitution experiments between modules. Specifically, we performed the following experiments individually: (**1**). Removal of the SFE module only, known as the MEVD-S model. (**2**). Replacement of the Transformer-CNN dual-branch encoder module with a CNN model, called the MEVD-T model. (**3**). Removal of the DRSN module only, named MEVD-D model. **Table 5** shows the experimental results for these different modules. By comparing these results, we can determine the impact of the different modules on the model performance.

Table 5. Model comparison with different components.

Method	Vulnerability	Acc (%)	Pre (%)	Rec (%)	F1 (%)
MEVD-S	Reentrancy	88.71	84.41	83.59	85.94
	Timestamp Dependency	85.52	83.01	89.32	86.04
	Infinite Loop	82.15	80.37	81.86	82.18
MEVD-T	Reentrancy	84.31	81.93	82.29	80.07
	Timestamp Dependency	82.24	80.46	85.17	82.75
	Infinite Loop	76.18	75.42	78.25	77.92
MEVD-D	Reentrancy	89.31	86.24	85.13	84.27
	Timestamp Dependency	85.34	83.82	87.58	85.67
	Infinite Loop	81.28	78.32	83.53	82.30
MEVD	Reentrancy	**92.13**	**89.49**	**90.15**	**89.28**
	Timestamp Dependency	**90.85**	**89.17**	**92.98**	**91.04**
	Infinite Loop	**86.94**	**84.15**	**85.63**	**86.21**

By analysis of the experimental results in Table 5, we come to a clear conclusion: when various modules are removed, the experimental performance of all models generally experiences a decrease. Particularly noteworthy is the MEVD-T model, which shows a significant decrease in F1 score of 8.29% compared to the MEVD model for timestamp dependency vulnerability detection. In the case

 doi:10.20944/preprints202312.2325.v1

of reentrancy vulnerability detection, the accuracy of the MEVD-T model is only 84.31%, a decrease of 7.82% compared to the MEVD model, with other evaluation metrics also falling short of the MEVD model. This phenomenon is due to the limitation of the CNN models in adequately capturing the code relationships, thus failing to extract vulnerability features effectively. In contrast, our constructed Transformer CNN dual-branch encoder module comprehensively extracts code global structure and fine-grained syntax semantic features, and fuses them into multi-scale feature vectors. This enables the model to learn vulnerability patterns, significantly improving detection performance. Further analysis of the MEVD-S and MEVD-D experimental results shows that the SFE and DRSN modules enhance semantic contextual details and effectively eliminate redundant information from feature vectors. For example, in reentrancy vulnerability detection, the SFE module improves accuracy by 3.42%, while in infinite loop vulnerability, the DRSN module improves F1 score by 3.91%.

6. Conclusions

In this paper, we propose a novel Multi-Scale Encoder Vulnerability Detection (MEVD) approach, which, in contrast to existing approaches, can enrich the semantic information of code features, fully extracting both global structure and local detail features of smart contract code, while effectively eliminating redundant elements within code features. In addition, we have introduced the concept of a Vulnerability Syntax-Driven Slice (VSS), which removes code statements unrelated to vulnerability characteristics, contains rich syntactic and semantic information, and preserves the data and control dependencies between statements. This helps the model focus on the critical characteristics of the vulnerability, capturing more features about the vulnerability. A large number of experiments have shown that MEVD outperforms traditional methods and state-of-the-art deep learning approaches in detecting vulnerabilities in smart contracts. For future work, we plan to combine deep learning techniques with large language model technology to further explore more effective methods for detecting vulnerabilities in smart contracts.

Author Contributions: Conceptualization, L.L. and J.G.; methodology, L.L. and J.G.; software, L.L.; validation, J.L., J.G. and L.L.; formal analysis, L.L.; investigation, J.G.; resources, L.L.; data curation, L.L.; writing—original draft preparation, L.L.; writing—review and editing, J.G. and L.L.; funding acquisition, J.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Key Research and Development Project in Shaanxi Province of China (No. 2022GY048), and the Scientific Research Program Funded by the Shaanxi Provincial Education Department (No. 19JC021).

Data Availability Statement: The experimental data used in this study will be posted on the repository (https://github.com/COPELONG/MEVD, (accessed on 22 December 2023)).

Acknowledgments: The authors are very grateful to the anonymous reviewers for their constructive comments, which helped to improve the quality of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alharby, M.; Moorsel, A.V. Blockchain Based Smart Contracts : A Systematic Mapping Study. In Proceedings of the Computer Science & Information Technology (CS & IT); Academy & Industry Research Collaboration Center (AIRCC), August 26 2017; pp. 125–140.
2. Gupta, R.; Tanwar, S.; Al-Turjman, F.; Italiya, P.; Nauman, A.; Kim, S.W. Smart Contract Privacy Protection Using AI in Cyber-Physical Systems: Tools, Techniques and Challenges. *IEEE Access* **2020**, *8*, 24746–24772, doi:10.1109/ACCESS.2020.2970576.
3. Wang, S.; Ouyang, L.; Yuan, Y.; Ni, X.; Han, X.; Wang, F.-Y. Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends. *IEEE Trans. Syst. Man Cybern, Syst.* **2019**, *49*, 2266–2277, doi:10.1109/TSMC.2019.2895123.

4. Khan, S.N.; Loukil, F.; Ghedira-Guegan, C.; Benkhelifa, E.; Bani-Hani, A. Blockchain Smart Contracts: Applications, Challenges, and Future Trends. *Peer-to-Peer Netw. Appl.* **2021**, *14*, 2901–2925, doi:10.1007/s12083-021-01127-0.
5. Wu, H.; Dong, H.; He, Y.; Duan, Q. Smart Contract Vulnerability Detection Based on Hybrid Attention Mechanism Model. *Applied Sciences* **2023**, *13*, 770, doi:10.3390/app13020770.
6. Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *Journal of Cases on Information Technology (JCIT)* **2019**, *21*, 19–32.
7. Almakhour, M.; Sliman, L.; Samhat, A.E.; Mellouk, A. Verification of Smart Contracts: A Survey. *Pervasive and Mobile Computing* **2020**, *67*, 101227, doi:10.1016/j.pmcj.2020.101227.
8. He, D.; Deng, Z.; Zhang, Y.; Chan, S.; Cheng, Y.; Guizani, N. Smart Contract Vulnerability Analysis and Security Audit. *IEEE Network* **2020**, *34*, 276–282, doi:10.1109/MNET.001.1900656.
9. Chu, H.; Zhang, P.; Dong, H.; Xiao, Y.; Ji, S.; Li, W. A Survey on Smart Contract Vulnerabilities: Data Sources, Detection and Repair. *Information and Software Technology* **2023**, *159*, 107221, doi:10.1016/j.infsof.2023.107221.
10. Qian, P.; Liu, Z.; He, Q.; Zimmermann, R.; Wang, X. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. *IEEE Access* **2020**, *8*, 19685–19695, doi:10.1109/ACCESS.2020.2969429.
11. Cai, J.; Li, B.; Zhang, J.; Sun, X.; Chen, B. Combine Sliced Joint Graph with Graph Neural Networks for Smart Contract Vulnerability Detection. *Journal of Systems and Software* **2023**, *195*, 111550, doi:10.1016/j.jss.2022.111550.
12. Zhang, L.; Chen, W.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. CBGRU: A Detection Method of Smart Contract Vulnerability Based on a Hybrid Model. *Sensors* **2022**, *22*, 3577, doi:10.3390/s22093577.
13. Wu, H.; Zhang, Z.; Wang, S.; Lei, Y.; Lin, B.; Qin, Y.; Zhang, H.; Mao, X. Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-Training Techniques. In Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE); IEEE: Wuhan, China, October 2021; pp. 378–389.
14. Li, M.; Ren, X.; Fu, H.; Li, Z.; Sun, J. ConvMHSA-SCVD: Enhancing Smart Contract Vulnerability Detection through a Knowledge-Driven and Data-Driven Framework. In Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE); IEEE: Florence, Italy, October 9 2023; pp. 578–589.
15. Yu, L.; Lu, J.; Liu, X.; Yang, L.; Zhang, F.; Ma, J. PSCVFinder: A Prompt-Tuning Based Framework for Smart Contract Vulnerability Detection. In Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE); IEEE: Florence, Italy, October 9 2023; pp. 556–567.
16. Atzei, N.; Bartoletti, M.; Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (Sok). In Proceedings of the Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6; Springer, 2017; pp. 164–186.
17. Liu, Z.; Qian, P.; Wang, X.; Zhuang, Y.; Qiu, L.; Wang, X. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Trans. Knowl. Data Eng.* **2021**, 1–1, doi:10.1109/TKDE.2021.3095196.
18. Jiang, B.; Liu, Y.; Chan, W.K. Contractfuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In Proceedings of the Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; 2018; pp. 259–269.
19. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Buenzli, F.; Vechev, M. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the Proceedings of the 2018 ACM SIGSAC conference on computer and communications security; 2018; pp. 67–82.
20. Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; Hobor, A. Making Smart Contracts Smarter. In Proceedings of the Proceedings of the 2016 ACM SIGSAC conference on computer and communications security; 2016; pp. 254–269.
21. Prechtel, D.; Groß, T.; Müller, T. Evaluating Spread of 'Gasless Send'in Ethereum Smart Contracts. In Proceedings of the 2019 10th IFIP international conference on new technologies, mobility and security (NTMS); IEEE, 2019; pp. 1–6.

22. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB); IEEE, 2019; pp. 8–15.
23. Yu, X.; Zhao, H.; Hou, B.; Ying, Z.; Wu, B. DeeSCVHunter: A Deep Learning-Based Framework for Smart Contract Vulnerability Detection. In Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN); IEEE: Shenzhen, China, July 18 2021; pp. 1–8.
24. Liu, Z.; Qian, P.; Wang, X.; Zhu, L.; He, Q.; Ji, S. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. *arXiv* 2021, arXiv:2106.09282.
25. Duy, P.T.; Khoa, N.H.; Quyen, N.H.; Trinh, L.C.; Kien, V.T.; Hoang, T.M.; Pham, V.-H. VulnSense: Efficient Vulnerability Detection in Ethereum Smart Contracts by Multimodal Learning with Graph Neural Network and Language Model. *arXiv* 2023, arXiv:2309.08474.
26. Sak, H.; Senior, A.W.; Beaufays, F. Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. **2014**.
27. Kipf, T.N.; Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv* 2016, arXiv: 1609.02907.
28. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, \Lukasz; Polosukhin, I. Attention Is All You Need. *Advances in neural information processing systems* **2017**, *30*.
29. Zamir, S.W.; Arora, A.; Khan, S.; Hayat, M.; Khan, F.S.; Yang, M.-H. Restormer: Efficient Transformer for High-Resolution Image Restoration. In Proceedings of the 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR); IEEE: New Orleans, LA, USA, June 2022; pp. 5718–5729.
30. Ding, X.; Zhang, X.; Ma, N.; Han, J.; Ding, G.; Sun, J. RepVGG: Making VGG-Style ConvNets Great Again. In Proceedings of the 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR); IEEE: Nashville, TN, USA, June 2021; pp. 13728–13737.
31. Zhao, M.; Zhong, S.; Fu, X.; Tang, B.; Pecht, M. Deep Residual Shrinkage Networks for Fault Diagnosis. *IEEE Transactions on Industrial Informatics* **2019**, *16*, 4681–4690.
32. Zhao, Z.; Bai, H.; Zhang, J.; Zhang, Y.; Xu, S.; Lin, Z.; Timofte, R.; Van Gool, L. CDDFuse: Correlation-Driven Dual-Branch Feature Decomposition for Multi-Modality Image Fusion. In Proceedings of the 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR); IEEE: Vancouver, BC, Canada, June 2023; pp. 5906–5916.
33. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed Representations of Words and Phrases and Their Compositionality. *Advances in neural information processing systems* **2013**, *26*.
34. Zhang, L.; Li, Y.; Jin, T.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. SPCBIG-EC: A Robust Serial Hybrid Model for Smart Contract Vulnerability Detection. *Sensors* **2022**, *22*, 4621, doi:10.3390/s22124621.
35. Durieux, T.; Ferreira, J.F.; Abreu, R.; Cruz, P. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In Proceedings of the Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering; June 27 2020; pp. 530–541.
36. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart Contract Vulnerability Detection Using Graph Neural Network. In Proceedings of the Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence; International Joint Conferences on Artificial Intelligence Organization: Yokohama, Japan, July 2020; pp. 3283–3290.