Smart Contract Vulnerability Detection Based on Hybrid Attention Mechanism Model

Huaiguang Wu 1, Hanjie Dong 1,*, Yaqiong He 1 and Qianheng Duan 2

1 College of Computer and Communication Engineering, Zhengzhou University of Light Industry, Zhengzhou 450001, China
2 Henan Key Laboratory of Network Cryptography Technology, Zhengzhou 450001, China
* Correspondence: donghanjie@email.zzuli.edu.cn

Abstract: A smart contract, as an important part of blockchain technology, has attracted considerable interest from both industry and academia. It provides the basis for the realization of a variety of practical blockchain applications and plays a crucial role in the blockchain ecosystem. While it also holds a large number of digital assets, the frequent occurrence of smart contract vulnerabilities have caused huge economic losses and destroyed the blockchain-based credit system. Currently, the security and reliability of smart contracts have become a new focus of research, and there are a number of smart contract vulnerability detection methods, such as traditional detection tools based on static or dynamic analysis. However, most of them often rely on expert rules, and therefore have poor scalability and high false negative and false positive rates. Recent deep learning methods alleviate this issue, but without considering the semantic information and context of source code. To this end, we propose a hybrid attention mechanism (HAM) model to detect security vulnerabilities in smart contracts. We extract code fragments from the source code, which focus on key points of vulnerability. We conduct extensive experiments on two public smart contract datasets (a total of 24,957 contracts). Empirical results show remarkable accuracy improvement over the state-of-the-art methods on five kinds of vulnerabilities, where the detection accuracy could achieve 93.36%, 80.85%, 82.56%, 85.62%, and 82.19% for reentrancy, arithmetic vulnerability, unchecked return value, timestamp dependency, and tx.origin, respectively.

Keywords: smart contract; vulnerability detection; deep learning; hybrid attention mechanism

1. Introduction

In recent years, new digital currencies have developed rapidly, among which the value and social influence of digital currencies represented by Bitcoin are inestimable [1]. In 2008, Satoshi Nakamoto [2] put forward the concept of blockchain for the first time in the Bitcoin white paper. Since then, blockchain has gradually come into people’s view. In 2015, the development of blockchain ushered in a big outbreak. Subsequently, blockchain was gradually applied in many traditional fields such as finance [3], supply chain management [4,5], transportation [6,7], medical care [8–11], and the Internet of Things [12–14].

Smart Contract [15] refers to a special program running on blockchain technology, which makes use of the consensus mechanism of blockchain to enable all participants to automatically reach an agreement without relying on the trust of a third party. The concept of smart contracts was first proposed by Nick Szabo and was initially just an idea without the right conditions for implementation. Blockchain is a distributed ledger technology realized by a consensus mechanism in a peer-to-peer network. Combined with blockchain technology, smart contracts have obtained an ideal execution environment, from concept to application. Ethereum [16] was the first blockchain platform to support smart contracts. In Ethereum, smart contracts exist in the form of contract accounts to manage electronic cryptocurrencies stored on the blockchain platform. By 2022, the value of digital assets managed by smart contracts in Ethereum alone exceeded $300 billion [17].
With the wide application of the smart contract, security issues of the smart contract are gradually being exposed. If there are vulnerabilities in the smart contract, the attackers can use them to steal electronic cryptocurrency managed by the smart contract or make it impossible to withdraw, which brings significant economic losses to the owners and participants of the contract. For instance, in 2016, the Decentralized Autonomic Organization (DAO) [18], the world’s largest crowdfunding project deployed on Ethereum, was hacked, causing more than 3 million Ether to leave the DAO resource pool. In the end, Ethereum adopted a hard fork strategy, which damaged the fairness of blockchain. In 2017, a security vulnerability occurred in Parity Multi-Sig Wallet contract [19], resulting in the embezzlement of more than 150,000 ETH. In 2021, the Automated Market Maker (AMM) contract [20] on the Binance Smart Chain had a vulnerability during the migration process and was exploited by hackers, leading to the theft of $50 million. It is thus important to check and ensure the security of smart contracts before deploying them to the blockchain.

The reasons why smart contracts are susceptible to security vulnerabilities can be summarized in the following aspects. Current programming languages and tools for smart contracts (e.g., Solidity) are still novel and crude. Moreover, the writing of smart contracts is different from traditional programming to some extent, and the lack of security awareness of developers will easily cause some potential vulnerabilities. Generally, traditional software can release multiple versions successively to update and iterate, continuously repair the discover vulnerabilities, and reduce losses. On the contrary, given the immutable characteristics of blockchain technology, smart contracts cannot be changed once deployed on the chain, even if the vulnerability is found to be exploited. On the one hand, smart contracts hold and control digital currencies (such as Bitcoin and Ether) on blockchain platforms making them attractive for malicious attackers. On the other hand, smart contracts are usually exposed to the open network environment, which further gives much incentive to hackers. They always explore and exploit the possible vulnerabilities in smart contracts to steal funds or hinder the normal operation of the blockchain network. Further, as the number of smart contracts explodes, it is increasingly important to check potential vulnerabilities of smart contracts more accurately and efficiently, before they are deployed.

Disadvantages of traditional approaches. From the perspective of whether to execute the program, traditional methods on finding vulnerabilities mainly can be divided into two categories, namely static analysis [21–23] and dynamic execution technology [24,25]. Although they have achieved good results, but they still have some problems. First, the existing methods rely heavily on hard rules (or patterns) defined by human experts. As the number of smart contracts increases rapidly, it is unrealistic to expect that a few experts can audit all the contracts and design the corresponding rules or patterns. What is more, expert rules are not always reliable, and some complex patterns are difficult to consider.

More recently, efforts have been made towards applying deep learning-driven techniques to vulnerability detection of smart contract. With its powerful feature mining ability, deep learning technology can extract implied features from the data itself and has extremely high universality. Tann et al. [26] introduces an LSTM model to detect smart contract security threats at an opcode level. Qian et al. [27] proposes a sequential model to focus on the reentrancy vulnerability, while Zhuang et al. [28] build the control flow graph to represent a smart contract function. However, they methods still have some shortcomings. They just regard the source code or opcode code as a text sequence, and ignore the key variables of data flow. This can lead to inadequate semantic modeling and unsatisfactory vulnerability prediction results.

To address this issue, we introduce a method of source code vulnerability detection based on the hybrid attention mechanism (HAM) model, which consists of consists of a single-head attention encoder combined with a bidirectional gated recurrent unit (Bi-GRU) and a multi-head attention encoder. Specially, we extract code fragments with key points of vulnerability from smart contact to improve the performance of deep learning models. Then the generated code fragments are converted into fixed-dimension vector. Next, the vector is fed into a hybrid attention mechanism network for feature learning. Finally, the single-head
attention features and the multi-head features are incorporated to produce the final vulnerability detection results. We conducted experiments on two public real datasets, which contains five types of vulnerabilities (i.e., reentrancy, timestamp dependency, unchecked return value, and arithmetic vulnerability).

In summary, the main contributions of this work are as follows:

- To our knowledge, we are the first to present the idea of fusing the single-head feature and the multi-head attention feature for smart contract vulnerability detection.
- We propose to extract code fragments with richer data flow and control flow information, which obviously enhance the detection performance of deep learning models.
- We compare our model with other ten state-of-the-art methods and the comparison results show that our model performs best in the detection of five types (reentrancy, timestamp dependency, unchecked return value, tx.origin, and arithmetic vulnerability) of smart contract vulnerabilities.

The rest of the paper is organized as follows. Section 2 introduces some background of Ethereum and smart contract, five types of vulnerabilities we aim to detect, and the motivation of our approach. Section 3 describes the related work about the vulnerability detection of smart contract. Section 4 presents the approach details. In Section 5, we show the experimental results. Section 6 concludes the paper.

2. Background

2.1. Ethereum and Smart Contract

As a representative of blockchain technology, Bitcoin can execute simple decentralized transaction scripts, but cannot run Turing-complete codes, so it is not considered a Turing-complete smart contract platform [29]. Ethereum [30], regarded as blockchain 2.0, supports the operation of Turing-complete smart contracts. It is the first and most popular smart contract platform to support smart contract operation. With the emergence of Ethereum, more and more blockchain platforms have begun to support the execution of Turing complete smart contracts, such as EOS [31], NEO [32], and Fabric [33].

In contrast to the Bitcoin system, the Ethereum platform not only leverages blockchain technology to maintain decentralized payments, but also enables users to develop smart contracts according to their needs. A smart contract is a piece of program code that runs on the blockchain. Any user can write a smart contract and deploy it to the blockchain. This process is irreversible. Users can call functions in the contract through external accounts to realize the interaction with Ethereum. Furthermore, smart contracts can complete transactions by sending messages to other contracts or calling functions. After the transaction verification of the entire network node is passed, the state changes are recorded in the block.

Ethereum supports smart contract programming languages such as Solidity [34], LLL, Serpent, and Vyper, but the vast majority of contracts are written in Solidity. The syntax of Solidity is similar to Javascript, which supports sophisticated features such as inheritance, library functions, user-defined types, and so on. Compared with traditional contracts, smart contracts have obvious advantages. First, smart contracts can automatically execute when certain preconditions are met without the need for intermediaries, which greatly saves costs; second, smart contracts enable direct transactions between users, thus speeding up the process of contract verification and execution; Third, smart contract has a copy on each device connected to the Internet, so that the execution process will not break the circuit.

2.2. Security Vulnerability in Smart Contracts

As a decentralized application running on the blockchain, the security threats of smart contracts are closely related to their operating environment. Smart contracts are written in a high-level language, then compiled into bytecode, and driven by blockchain transactions, running on a virtual machine with blockchain as the storage basis. During the whole process, different security threats will be faced and accompanied by the discovery of security vulnerabilities. In 2017, Atzei [35] et al. analyzed the security vulnerabilities of Ethereum...
smart contract, and for the first time classified the vulnerabilities of smart contract from the three levels of programming language, virtual machine and blockchain. In the same year, Dika [36] followed the classification of Atzei et al., categorized known security problems and assigned safety levels (1-3 marks low, medium and high). Dingman [37] proposed a NIST framework to check the smart contract code and report 49 bugs categorized into 10 classes. Through the investigation of contract vulnerabilities and related work, we select five common vulnerabilities as research object, namely Integer Underflow vulnerability, Integer Overflow vulnerability, Unchecked Return Value, Transaction ordering dependence, Timestamp Dependency and Reentrancy vulnerability, among which Integer Underflow vulnerability, Integer Overflow Vulnerability are counted as arithmetic vulnerabilities.

2.2. Reentrancy

Reentrancy is a type of security vulnerability that is extremely harmful. This vulnerability usually occurs in the process of transferring funds to an external user address and is caused by the external user address recursively calling the same function of the contract. There is a function without a function name and parameters in the solidity contract, which is called the fallback function. This function is automatically triggered when receiving a transfer. Before the transfer operation modifies the contract’s storage state variable, it may be subject to a reentrancy attack, because the attacker can inject a malicious fallback function to call the same function of the contract again. In the theDAO incident, the attacker used the reentrancy vulnerability to maliciously repeat the transfer to complete the attack. To address this issue, it is better to utilize the `send()` or `transfer()` when performing transfer operations, and change its internal state before using the low-level call function.

Figure 1 is a withdrawal function with a reentrancy vulnerability. This function first checks whether the caller has enough balance, then sends ether through the call statement, and finally deducts the corresponding balance of the caller. If another contract C calls the withdrawal function, the call statement will trigger the fallback function of contract C and hinders until the callback function of C finishes execution. At this time, the balance of the caller C has not been deducted. If C’s fallback function calls the withdrawal function again, the detection of Line 2 can still pass, and finally C can withdraw the ether that did not belong to it.

```solidity
function Cashout(uint_amount) public {
    require(balances[msg.sender] >= _amount;
    msg.sender.call.value(_amount)();
    balances[msg.sender] -= _amount;
}
```

Figure 1. Reentrancy.

2.2.2. Arithmetic Vulnerability

Integer overflow and integer underflow are a common type of error in many programming languages. Specifically in smart contracts, integer calculations are often closely related to the transfer of digital assets, which may cause very serious consequences. Integer overflow mainly occurs in the coin issuing contracts, and there are endless cases of attacks because of that in real contracts, which is a high-risk vulnerability. Taking Ethereum as an example, the unsigned integer in Solidity will be automatically truncated if the calculation result exceeds the range of representation during the calculation process, and the high-order numbers will be discarded, resulting in abnormal calculation results. As shown in Figure 2, the amount in line 2 is the multiplication of two unit 256 values. If a very large
value causes the calculation result to overflow and the value of amount becomes 0, then the check logic of require_value > 0&&balance[msg.sender] ≥ amount in line 3 will fail. This is because the amount should normally be a non-zero value. Attackers can exploit this vulnerability to transfer a large amount of tokens to an address with a small amount of tokens, causing serious economic losses. For the problem of integer overflow, Ethereum provides the SafeMath library. The numerical calculation operations in the library will detect integer overflow behavior and throw an exception when overflow occurs. Using the SafeMath library can effectively prevent integer overflow attacks. Nevertheless, irregular programming style and ignorance of integer overflow vulnerabilities still lead to frequent occurrence of integer overflow attacks.

```solidity
function paidout(address[] _buyer, uint256 _value){
    uint256 amount=uint256(_buyer.length)*_value;
    require(_value>0&&balances[msg.sender]>=amount);
    ...
    return true;
}
```

Figure 2. Arithmetic Vulnerability.

2.2.3. Tx.origin

On blockchain, a block consists of a set of transactions, and the state of the blockchain is updated several times within each transaction. The state of a smart contract is determined by the values of its fields and the current balance. In most cases, when a user initiates a transaction to call a smart contract in the network, there is no guarantee that the transaction will run in the same state of the contract as when the transaction was initialized. In fact, when a smart contract is invoked by a user’s transaction, the actual state of the smart contract is undetermined.

For example, if user 1 and user 2 respectively send transactions \(T_i\) and \(T_j\) to the smart contract at the same time \(t\), neither user knows which transaction will run first. Even if user 1 sends transaction \(T_i\), \(T_i\) before user 2 sends \(T_j\), \(T_i\) is not guaranteed to run before \(T_j\). If \(T_i\) is executed first, it will change the contract state \(S \rightarrow S_i\); but if \(T_j\) is executed first, it will change the contract state \(S \rightarrow S_j\). Therefore, the final state of the contract depends on the order in which transactions are executed, and the order of these transactions is determined only by the miners of the block. Contracts that use the tx.origin variable to authenticate users are often vulnerable to phishing attacks, which can trick users into performing authentication operations on vulnerable contracts. Figure 3 is a code example of a tx.origin vulnerability.

```solidity
function transfer(address test uint256 amount){
    if(tx.origin!=owner) throw;
    test.send(amount);
}
```

Figure 3. Tx.origin.
2.2.4. Timestamp Dependency

A timestamp is a sequence of time characters that uniquely identifies a certain moment. Timestamp dependence in Ethereum means that the execution of smart contracts depends on the timestamp of the current block. With the different timestamps, the execution results of the contract are also different. When a smart contract utilizes block variables (e.g., BLOCKHASH, TIMESTAMP, NUMBER) as a call condition to perform some key operations such as sending cryptocurrency or as a seed to generate a random number, the miner can set the timestamp of the block as he wishes, within a time interval of 900 s. In this way, hackers can influence the execution of contract functions by controlling the time stamp, especially the money transfer in some contracts is based on the time stamp. Figure 4 is an example of a lottery smart contract written in solidity language, and there is a time stamp dependency vulnerability on line 4 of the code.

```solidity
function Lottery() public{
    if(msg.sender==owner)
        uint sum=0;
        uint WinNum=uint(block.blockhash(block.number-1))%TestsAccounts+1;
        ...
        return;
}
```

Figure 4. Timestamp Dependency.

2.2.5. Unchecked Return Value

Unchecked return value is a kind of vulnerability caused by the failure to check the execution of the transfer operation in the smart contract code. Solidity implements money transfer through three functions: transfer(), send(), and call(). When the transfer fails, transfer() automatically throws an exception, yet the send() and call() functions do not and continue to execute the remaining code. If the execution of the transfer function is not checked, there is an opportunity for an attacker to deliberately make the transfer fail in some way, thereby affecting the normal execution of the contract.

As shown in Figure 5, the function of pay() is that the contract pays a certain amount of tokens to the account whose address is address and sets the paid status to true. address.send(amount) on line 3 and the code at line 5 do the above. Suppose that the account at address is an attacker at this point, and that the malicious account somehow causes line 3 to fail to return false and proceed to the next line of code. In fact, the paid state is still rue, and the pay function is not consistent with the expected execution state. Therefore, it is safer to write require(address.send(amount)) in line 4. Only when line 4 is successfully executed will the next line of code be executed, avoiding the vulnerability of unchecked return value.

```solidity
function pay(uint256 amount){
    require(!paid);
    address.send(amount);//require(address.send(amount))
    paid=true;
}
```

Figure 5. Unchecked Return Value.
2.3. Source Code Vulnerability Detection

In this paper, we represent the source code as vectors rather than bytecode and opcode. The reasons for this are two-fold. Different from the source code written by high-level language, bytecode is completely represented by binary of 0 and 1, which is unreadable and difficult to directly extract the syntax structure of code or the code sequence information corresponding to the functions. As for opcode, its length is not fixed and contains invalid information. In the opcode sequence obtained after disassembly, each opcode has a different function and structure, which leads to the uncertain length of the opcode. For most machine learning algorithms, data with variable length are difficult to learn directly as the input. At the same time, the opcode sequence also contains some noise information unrelated to the characteristics of the vulnerability, which will interfere with the matching of the subsequent vulnerability code. Therefore, we favor the source code with rich semantic information.

3. Related Work

In the past years, academia has proposed a number of different solutions for ensuring correctness and safety of smart contracts. These mainly include attempts on symbol execution, fuzzy testing, formal verification, program analysis, and machine learning.

3.1. Symbol Execution

Symbolic execution [38] symbolizes the code variables and replaces the specific value input with the symbolic input to realize the analysis of the program. Oyente [21] is the first smart contract vulnerability detection tool based on symbolic execution, which can detect smart contract transaction sequence dependency, timestamp dependency, reentrancy, and other vulnerabilities, and output the corresponding symbolic paths. The results of Oyente vulnerability detection show that the code coverage is low, and there is a high false positive rate and false negative rate in the detection of integer overflow vulnerability.

The MAIAN [22] analysis tool completes the detection task in two steps. It first enters the contract bytecode and analysis strategy into the symbol analysis component and executes all possible execution paths for each candidate contract symbolically until it finds and marks a path that satisfies a set of predetermined properties, and then executes the contract to verify that the results are consistent. Osiris [23] is an improvement on the basis of Oyente, which aims at detecting integer overflow vulnerabilities in smart contracts, combining symbolic execution and taint analysis techniques. The tool takes as input the smart contract’s bytecode or Solidity source code, which is further compiled into bytecode for analysis and outputs the integer overflow information that exists in the contract. The vulnerability detection function consists of three modules, i.e., symbolic execution, taint analysis, and integer overflow detection. On top of that, Securify [39] and Slither [40] are also Ethereum smart contract security analysis tools based on symbolic execution technology.

3.2. Fuzzing Testing

Fuzzing technology is a dynamic software security detection technology. It can judge whether the program has potential security risks by constructing random test cases and monitoring whether the program will throw exceptions such as memory corruption. ContractFuzzer [24] is the first fuzzing framework based on the Ethereum platform for smart contract security vulnerability. The tool includes an offline EVM instrumentation tool and an online fuzzing tool. It takes the test case generated by the smart contract API specification as input, obtains the execution log file, and implements vulnerability detection by analyzing the execution log. ILF combines deep learning with fuzzing [41] to generate better test cases through neural networks, thereby enabling vulnerability detection. sFuzz [42] is based on AFL and adopts a feedback adaptive fuzzing strategy to improve the coverage of test cases, so as to achieve the purpose of detection. Although existing fuzzing tools can better detect contract vulnerabilities, the test samples generated by them are relatively random, resulting in insufficient code coverage and lower efficiency.
3.3. Formal Verification

Formal verification is one of the most accurate methods for verifying system accuracy and also one of the earliest methods to check the behavior of smart contracts. It adopts logical language to formally model smart contracts, and proves the correctness of smart contract functions and runtime security through rigorous mathematical reasoning. In 2016, Bai et al. [43] proposed a formal verification framework for smart contracts, which adopted the Promela modeling language to model the smart shopping contract and used the SPIN tool to perform model testing. However, this method was limited to a single contract, excluding the contract interaction process and complex contracts. S. Amani et al. proposed the EVM formal model of smart contracts and leveraged the Isabelle/HOL proof tool [44] to define the EVM formal model in detail and verified the security of the model. E. Hildenbrandt et al. [45] presented the formal semantic framework KEVM, which provides an executable and readable formal model of complete EVM semantics, including all EVM instructions. Bhargavan et al. [46] outlined a framework for analyzing and verifying the security of smart contracts using the F* functional programming language. The source code of the contract and the EVM bytecode are compiled into F* language, and the security and functional correctness of the real-time runtime are verified respectively. VerX [47] is the first automated verifier able to automatically prove functional properties of Ethereum smart contracts. Additionally, the delayed predicate approach is the key to achieving automatic verification of smart contracts, which is a new symbolic engine that can soundly handle Ethereum features. Although formal verification technology can understand the code at a higher semantic level, the automation of this vulnerability detection method is relatively low, requiring experienced researchers to participate in the verification work, which increases the detection cost.

3.4. Program Analysis

Program analysis and taint analysis techniques are also applied to the vulnerability detection of smart contracts. Taint analysis usually takes the external input as taint data and tracks the flow of information related to taint data to determine whether it will affect the operation of the program, so as to explore the vulnerability. Slither [40] and SmartCheck [48] are tools based on program analysis. Slither converts the contract source code into an intermediate representation of SlithIR, and then exploits the data flow and taint tracking technology for vulnerability detection. While SmartCheck converts the source code into an XML intermediate representation and then utilizes XPath patterns to detect smart contract vulnerabilities.

3.5. Machine Learning

In recent years, machine learning has achieved great success in the field of security. Machine learning has high automation and can effectively improve detection efficiency. Qian et al. introduced BiLSTM model [27] and the attention mechanism to detect reentrancy vulnerabilities by capturing control flow information and semantic information in contracts. ContractWard chooses contract opcodes as the model input, and adopts a variety of machine learning algorithms to detect contract vulnerabilities. DeeSCV/Hunter [49] proposed the concept of vulnerability candidate slices to capture key characteristics of vulnerabilities, focusing on reentrancy and time dependency. In the literature [50], the contract bytecode was first associated with RGB images, the contract bytecode was converted into fixed-size RGB encoded images, and the convolutional neural network was used for automatic feature extraction and learning to detect smart contract vulnerabilities. However, the interpretability of the contract bytecode represented by the RGB image is limited, and the vulnerability detection effect greatly depends on the performance of the neural network. Zhuang et al. explored the use of the graph neural network (GNN) to detect smart contract vulnerabilities [28]. They constructed a contract graph to represent the syntactic and semantic structures for a smart contract function and designed a degree-free graph convolutional neural network (DB-GCN) and a novel Temporal Messaging Network (TMP). This method
implements the contract graph generation separately for each vulnerability, but it is difficult to migrate and expand and the detection accuracy is limited. Gupta et al. [51] proposed a deep learning-based scheme to detect vulnerabilities at an opcode level. And they rate smart contract as safe/vulnerable based on probability value for rewarding or penalizing the users who deploy them. But they uses a simple binary vector with length equal to the size of machine instruction list to represent opcodes, there will be large spare vectors when the number of instructions increases.

4. Our Approach

In this section, we propose an approach to detecting contracts vulnerability for Solidity smart contracts based on hybrid attention mechanism. Our method takes source code as input and tells whether a vulnerability is present or not. The overview of the proposed model is illustrated in Figure 6, which consists of three phases: (1) the preprocess phase, which extracts code fragments from the source code with potential vulnerabilities; (2) the training phase, which put the word embedding matrix into our model for feature learning; and (3) the vulnerability detection phase, which utilizes the fully connected network to obtain the final classification results. In what follows, we give a detailed description of the three components one by one.

![Figure 6. The overall process of our approach.](image)

4.1. Code Fragments Extraction

In the field of software vulnerability detection, SySeVR [52] proposes to obtain relevant representations of program vulnerabilities using program-based syntax and semantics. In this paper, the smart contract dataset is processed according to the SySeVR method, and the key slice representation of the contract is extracted. Given that the dataset used in this paper is composed of real contracts on the Ethereum platform, which contain a large number of comment statements and blank statements. For the network model, it is necessary to control the data length of the input model to reduce the impact of irrelevant data on the model, so we delete these contents first. Considering that some code lines have nothing to do with vulnerabilities, we extract the key slice representation of the contract, i.e., code fragment to represent a smart contract. The so-called key slice is the “center of the vulnerability”, or a code fragment that implies the existence of a vulnerability. In this paper, we focus on five types of vulnerabilities. The potentially vulnerable features are represented in Table 1. With key slice as the center, code fragments can be divided and combined. The contract slicing can keep as much semantic information as possible with the least data length without affecting the performance of the model for vulnerability detection. In this way, useless codes and functions in the contract are removed, while retaining key syntax and semantic information of the contract. As a result, a code fragment is composed of multiple lines of code that are related to each other in terms of data-flow dependency or control-flow dependency. The data preprocessing process is shown in Figure 7.
Table 1. The vulnerable features corresponding to different vulnerabilities.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Vulnerable Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>call.value(), fallback()</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>tx.origin variable</td>
</tr>
<tr>
<td>Timestamp dependence</td>
<td>block.number, now, or block.timestamp</td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>send(), call()</td>
</tr>
<tr>
<td>Arithmetic vulnerability</td>
<td>integer variables</td>
</tr>
</tbody>
</table>

Figure 7. Extract code fragment.

When building the smart contract source code dataset, two distinct characteristics of the solidity source code are considered. One is the Solidity source code, whose main structure consists of contracts and functions, including solidity keywords, operators, delimiters, custom variables, etc.; second, the vulnerability may exist in any line of the entire code file. Therefore, according to feature one, it is first necessary to perform lexical analysis on the solidity source code and add a token to each word in the source code to identify the corresponding type. On the one hand, for general code statements that do not contain symbols, word segmentation is performed according to spaces. On the other hand, for code statements that contain delimiters, we distinguish words from delimiters. Among them, self-defined identifiers have no meaning. To remove the effect on the model, the custom identifier of the function name in each code snippet is replaced with FUN1, FUN2, FUN3, custom identifiers for variables with VAR1, VAR2, VAR3. The whole process is shown in Figure 8.

Figure 8. Symbol representation process.

4.2. Word Embedding Layer

After word segmentation, it cannot be directly used as the input of the deep learning model, so each word needs to be vectorized to convert it into numerical value. If a
code fragment contains vulnerability, it may be caused by the combined influence of several pieces of code before and after. Therefore, it is necessary to adopt a word vector representation method that can contain as much semantic information as possible.

Word Embedding is essentially a neural network, which can map each word in the text into a short vector of a unified dimension, quantitatively measure the semantic relationship between words, and mine potential connections between words. At present, some traditional word vector representation methods, such as TF-IDF, One-Hot, etc., are all aimed at the representation of a single word, which can’t represent the information between words. Moreover, this method generally causes the feature dimension to be too large, resulting in low computational efficiency. In response, scholars have developed a variety of new methods. Currently, mainstream methods of word embedding based on content text include Word2vec [53], Glove, Fasttext, CodeBERT, etc. Furthermore, graph-based methods contain Deepwalk, Node2vec, and GCN. In this paper, we choose the widely used Word Embedding technology, i.e., Word2vec to model the word vector. It takes text corpus as input and obtains word vectors through model training. Word2vec is composed of two language models, CBOW and Skip-Gram. The CBOW model predicts the central word through the context of the word, while the Skip-Gram model predicts the words in the context of the given central word. It is proved in [54] that skip-gram model may have higher accuracy than CBOW, we thus exploit the skip-gram in this paper. As a result, the disaster of dimension explosion is solved to a certain extent. In the model, \( \mathbf{W} = (w_1, w_2, \ldots, w_L) \) indicates \( L \) words in a contract or a code fragment. Since the number of each code fragment is different, we make some adjustments. If the length of code fragment exceeds the fixed dimension, we will remove the ending part of vector. Conversely, we will conduct zero-padding at the end of vector. Thus, each word \( w_i \) is converted into a fixed-dimension vector \( \mathbf{x}_t \). It can be formulated as follows:

\[
\mathbf{x}_t = \mathbf{W} \mathbf{h}_i
\]

where \( \mathbf{h}_i \) is the one-hot encoding vector of \( w_i \), \( \mathbf{W} \in \mathbb{R}^{d \times V} \), \( V \) is fixed-sized value and \( d \) is the dimension of word embedding vector. From this, we can obtain a one word embedding matrix \( \mathbf{x} = (x_1, x_2, \ldots, x_L) \).

4.3. Hybrid Attention Mechanism

The hybrid attention mechanism consists of two sub-modules, a single-head attention encoder and a multi-head attention encoder. The two encoders employ different attention mechanisms and feature extractors. Specifically, the single-head attention encoder first receives the word embedding representation of code fragments, inputs them into the bidirectional Gated Recurrent Neural Unit Network (BiGRU), and then leverages a single attention mechanism module to assign different weights to different words according to their importance. The use of the attention mechanism not only allows the model to automatically judge the important parts of the code, but also improves semantic interpretability. Further, for a certain word, a multi-head attention encoder is introduced in this paper to better encode it. The encoder can take into account the information of other words, combine single and multi-head attention encoder modules, and make full use of the advantages in RNN and Transformer models, thereby improving the performance of smart contract vulnerability detection.

4.3.1. Single-Head Attention Encoder

Recurrent neural network (RNN) is a kind of neural network, which is improved from feed forward neural network [55]. RNNs employ its internal state (memory) to process variable-length input sequences, but such a structure also has defects. The processing ability for long sequences is poor, and gradients are prone to disappear. In addition to that, the time and computational cost required for RNN training are enormous. To solve these problems, long short-term memory neural network (LSTM) [56] and gated recurrent unit neural network (GRU) [57] are proposed. Long short-term memory neural network is a
variant of RNN. Yet unlike RNNs, LSTMs have feedback connections. A common LSTM cell is composed of a cell, an input gate, an output gate, and a forget gate. Similar to LSTM, GRU is also a solution to the vanishing gradient in RNN. In comparison with LSTM, the experimental effect presented by GRU has almost no change, but the structure of GRU is simpler. Compared with LSTM, GRU has only two gate functions: update gate and reset gate, which new state information combined with the previous memory unit is determined by the reset gate, and which information of the previous memory unit needs to be stored in the current timestep is controlled by the update gate. The structure diagram of GRU model is shown in Figure 9.

Figure 9. The basic structure of Gated Recurrent Unit.

Where $z_t$ is update gate, $r_t$ means reset gate, $h_{t-1}$ indicates the status information of the previous moment, $h_t$ is current status information, and $\sigma$ represents the sigmoid function that converts data into a value in the range of 0 to 1 and acts as a gated signal. The specific calculation process are as follows:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

(2)

$$z_t = \sigma(W_z[h_{t-1}, x_t])$$

(3)

$$\tilde{z}_t = \sigma(W_\tilde{z}[h_{t-1}, x_t])$$

(4)

$$\tilde{h}_t = \tanh(W_h[r_t \times h_{t-1}, x_t])$$

(5)

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t$$

(6)

where $W_r, W_z, W_\tilde{z}$ are the parameters learned through training, and is the candidate hidden layer state, which is only related to the input $x_t$ and the hidden state $h_{t-1}$ of the previous layer.

The disadvantage of the traditional GRU neural network is that it can only read the input source code information from a single direction. However, the actual situation needs to fully consider the feature representation of the front and back source codes. To address this problem, this paper adopts BiGRU neural network to capture the vector features from the forward and the backward directions. The principle of BiGRU is shown in Figure 10.
The input vector can be expressed as:

\[ \begin{align*}
    \overrightarrow{h}_1 &= \text{GRU}(\overrightarrow{h}_{t-1}, x_t) \\
    \overleftarrow{h}_1 &= \text{GRU}(\overleftarrow{h}_{t-1}, x_t)
\end{align*} \]  

(7)  
(8)

After that, the concatenation operation is performed to obtain the final hide layer state of output, as shown in the following Equation.

\[ h_t = \overrightarrow{h}_t + \overleftarrow{h}_t \]  

(9)

The attention mechanism stems from the human brain’s ability to perceive new things, allocating more attention to what is important and less attention to what is not. By assigning different probability weights to the hidden layer units of neural networks with the help of the attention mechanism, more favorable features can be extracted. Attention connects two different parts by automatically weighing and transforming the data, highlighting the key points to show better performance. Due to the effectiveness of the attention mechanism in strengthening key information in natural language processing, this paper adds an attention mechanism to the BiGRU neural network to enhance the accuracy of the model.

In the Solidity source code, the contribution of different words to the vulnerability detection results is obviously different. Code fragments containing custom function calls are generally more important than keywords defined by the language itself. Based on the attention mechanism, we can distinguish the contributions of different words. The equation can be expressed as:

\[ u_t = \tanh \left( W_u \begin{bmatrix} \overrightarrow{h}_t \\ \overleftarrow{h}_t \end{bmatrix} + b_u \right) \]  

(10)

\[ a_t = \text{softmax} \left( u_t^T + u_w \right) \]  

(11)

\[ \sum_{i=1}^{L} a_i = 1 \]  

(12)

\[ V_1 = \sum_{i=1}^{L} a_i \begin{bmatrix} \overrightarrow{h}_i \\ \overleftarrow{h}_i \end{bmatrix} \]  

(13)
where \( W_{w}, b_{w} \) and \( U_{w} \) are parameters that need to be trained. \( \left[ h_{t}, h_{t} \right] \) is the merging result of two hidden layer states obtained in the previous stage, which is still a hidden layer vector. The following we input this vector into a single layer perceptron to obtain the intermediate state vector \( h_{t} \); then calculate the product of \( u_{t} \) and \( h_{w} \), we next adopt the softmax function to obtain the normalized contribution \( a_{t} \). Variable \( a_{t} \) represents the weight of \( x_{t} \) in the code fragment, which is a real value. Softmax refers to the normalized result of the tokens in all positions of \( 1 \sim L \), and the sum is equal to 1; finally, the product of the hidden layer state and weight of all words is accumulated to achieve the feature vector representation of a code fragment and finally composes a new feature vector \( V_{t} \).

### 4.3.2. Multi-Head Attention Encoder

Whether RNN, LSTM and GRU, or their variant structures, can only be calculated from left to right or from right to left in sequence when calculating the text. This mechanism will lead to two problems. For one thing, the calculation of timestep \( t \) depends on the calculation result at time \( t - 1 \), which will limit the parallel calculation ability of the model; for another, various gate mechanisms of LSTM and GRU can solve the issue of long sequence dependency to a certain extent, but the performance will drop sharply when encountering a particularly long sequence. In 2017, Google emphasized the role of attention mechanism and proposed the Transformer model.

Transformer is a Seq2Seq model [58], which mainly applies to complete tasks such as machine translation. The biggest difference from LSTM and GRU is that the Transformer model does not need to be processed sequentially according to the order of the words, and can train all the words at the same time, which greatly improves the degree of parallelism and increases computational efficiency. Moreover, Transformer introduces Positional Encoding to understand the position of each word in the sentence, not just the semantics of the word itself.

Figure 11 depicts the Transformer encoder. The input sequence first undergoes word embedding and position encoding to obtain a word embedding matrix \( X \) that contains position information. The two dimensions of \( X \) represent word embedding dimension and sequence length respectively. Feature analysis is then performed in parallel through eight attention heads, where each attention head has three weight matrices, namely \( W_{q}, W_{k}, W_{v} \). For the word vector \( x_{t} \), the Query vector \( q_{t} \), the Key vector \( k_{t} \), and the Value vector \( v_{t} \) can be obtained by calculating it with three weight matrices. After concatenating the three vectors of all words, the query matrix \( Q \), key matrix \( K \), and the value matrix \( V \) are, respectively, acquired. To obtain the attention weight of a certain word, the query vector \( q_{t} \) of the word multiplied by the key matrix \( V \) makes the scores of all words in the sequence for the current word. After dividing each score by the square root of the key vector dimension, use softmax to obtain the probability, then multiply it with the corresponding word value vector, and finally accumulate all the value vectors to obtain the output vector \( z_{t} \) of the current word.

![Figure 11. Transformer Encoder.](image-url)
The multi-head attention encoder in this paper no longer involves the recurrent neural network, but utilizes two steps to acquire the feature vector. The first step is to obtain the positional encoding information of each word. To capture the order of words in a code fragment, the feature $p_t$ of position embedding is introduced in this paper. It can add the position information of each word in the word vector. Then, we do linear transformation to the word vector $x_t$ and position embedding for each word. The specific process are expressed in the following equation.

$$p^j_t = \begin{cases} 
\sin\left(\frac{t}{10000^2i}\right) & \text{if } i = 2k \\
\cos\left(\frac{t}{10000^2i}\right) & \text{if } i = 2k+1 
\end{cases}$$ (14)

$$x_t = (x_t + p_t)W_p + b_p$$ (15)

where $t$ is the position of the word, $i$ is the dimension position in the word vector, and $d$ is the output dimension.

Self-Attention mechanism is called intra-attention, whose query vector ($Q$), key value ($K$) and value ($V$) all come from the same input, while the query vector in traditional attention mechanism generally comes from the outside. Self-Attention uses the attention mechanism to dynamically generate weights of different connections, so as to better learn distance dependence. The core of the self-attention mechanism lies in the dot product attention. Its calculation method is shown as follows.

$$Q_j = XW^q$$ (16)

$$K_j = XW^k$$ (17)

$$V_j = XW^v$$ (18)

$$\text{head}_j = \text{softmax}\left(\frac{Q_jK_j^T}{\sqrt{d_k}}\right)V_j$$ (19)

$$\text{MultiHead} = \text{concat}(\text{head}_1, \ldots, \text{head}_i, \ldots, \text{head}_m)$$ (20)

$$\text{MAT} = \text{Norm}(X + \text{MultiHead} \times W^v)$$ (21)

$$V_2 = \text{Maxpool}(\text{MAT})$$ (22)

where $X$ is the word embedding matrix that comprises of $x_t$. $W^q$, $W^k$, $W^v$ are the weight matrix to be trained, and $W^q \in \mathbb{R}^{d \times d_q}$, $W^k \in \mathbb{R}^{d \times d_k}$, $W^v \in \mathbb{R}^{d \times d_v}$. Then $m$ attention heads are connected in parallel. For each attention head, there is $d_q = d_k = d_v = \frac{d}{m}$, $W^o \in \mathbb{R}^{hd_v \times d}$. When $m$ head vectors are merged, the word vector representation $V_2$ is finally obtained after residual addition, normalization processing, and max pooling operation.

### 4.4. Vulnerability Classification Detection

After the feature extraction of $V_1$ and $V_2$, feature fusion is conducted through the concatenation method, thus we obtain a new $V_c$. Then, a single fully connected layer to the activation function named softmax, and the end outputs the predicted detection result in the form of probability. The specific formula is as follows:

$$V_c = \text{concat}(V_1, V_2)$$ (23)

$$\hat{y}_c = \text{softmax}(FC(V_c))$$ (24)

$$H(y_c, \hat{y}_c) = -\sum_c y_c \log \hat{y}_c$$ (25)

where $\hat{y}_c$ is the predicted probability, and $y_c$ it true probability.
5. Experiments and Results

In this section, we empirically evaluate our method on two public datasets, namely SmartWild and SBcurated. Our experiments are centered on answering the following three research questions (RQs):

RQ1: Can the representation method of contract fragments make our hybrid attention model do a better job? For answering this question, we conducted an experiment on the whole source code as well as code fragments with data dependency and control dependency, and compared their performance.

RQ2: How effective are our deep learning models when compared with other state-of-the-art vulnerability detection tools? For answering this question, we will compare our method with the other existing detection methods, mainly including a series of sequential models (i.e., RNN, GRU, LSTM, BiLSTM, BiGRU) and traditional formal analysis methods (i.e., Oyente, Mythril, Smartcheck).

RQ3: Can our approach identify multiple types of smart contract vulnerabilities, and how about the performance of different deep learning models? For answering this question, we will conduct experiments on four types of vulnerabilities (i.e., Transaction ordering dependency, Timestamp Dependency, and Reentrancy vulnerability).

5.1. Datasets and Processing

In recent years, the Ethereum community has developed plenty of tools to analyze vulnerable smart contracts, but there is no standardized dataset. To collect enough data for our experiment, we use the SmartWild dataset collected by Durieux et al. [59] and a manually constructed dataset Sbcurated. This dataset comes from smart contracts that have been traded in the real world with a total number of 47518. These contracts have been repeatedly screened and have potential vulnerabilities, using the real Ethereum contract address as a unique identifier. Since there are many smart contract compiler versions, and some lower compiler versions are rarely run on Ethereum. In order to better simulate the current smart contract situation, this paper filters out some smart contracts with premature compiler versions when using the dataset, and finally adopts 24,957 smart contracts among them for experiments.

The Sbcurated dataset consists of two parts, one of which is a real-world contract with vulnerabilities, and the other is a manually constructed contract with vulnerabilities. Smart contracts in this data set are manually marked with vulnerability locations and categories, which can be used to assess the effectiveness of smart contract analysis tools in identifying vulnerabilities. The dataset consists of 136 samples.

The research [59] shows that the combination of Mythril and Slither is both accurate and efficient, so we apply them to initially label the SmartWild dataset. For contracts with inconsistent tool detection results, the Remix tool was used for further manual verification. First, this paper compiles and deploys the smart contract to be audited on Remix, then manually calls the function that may cause the vulnerability, and tests whether the execution of the function will trigger the vulnerability through the input of the constructor, and finally obtains the result of whether the smart contract is vulnerable. A vulnerable contract is labelled as “1” or else “0”. Through the above operations, we can find that among the 24,957 smart contracts, 7851 contracts contain vulnerabilities. Among them, there were 1358 arithmetic overflow vulnerabilities, 1440 reentrancy vulnerabilities, 1927 timestamp dependency, 211 Unchecked Return Value and 1000 tx.origin. We randomly selected 80% of the samples as a training set, and the remaining 20% as the testing set, as shown in Table 2.
Table 2. Smart contract samples distribution.

<table>
<thead>
<tr>
<th>Category</th>
<th>Training Set</th>
<th>Testing Set</th>
<th>Total Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerable sample</td>
<td>6281</td>
<td>1570</td>
<td>7851</td>
</tr>
<tr>
<td>Non-vulnerable sample</td>
<td>13,685</td>
<td>3221</td>
<td>17,106</td>
</tr>
<tr>
<td>Arithmetic vulnerability</td>
<td>1086</td>
<td>272</td>
<td>1358</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>1152</td>
<td>288</td>
<td>1440</td>
</tr>
<tr>
<td>Timestamp dependency</td>
<td>1542</td>
<td>385</td>
<td>1927</td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>1696</td>
<td>423</td>
<td>2119</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>806</td>
<td>201</td>
<td>1007</td>
</tr>
</tbody>
</table>

For better experimental evaluation, we employ our proposed method exacting code fragments. A code fragment deals with multiple lines of code that contains key syntax and semantic information of the contract. Then we sift through each fragment and label the corresponding ground label as “1” (malicious) or “0” (benign). Furthermore, to focus on the key features of code fragments, we convert them into vector representations, which are acceptable to the deep learning models. First, we remove the non-ASCII character, blank lines, and comments. Then custom identifier of function is mapped as symbolic name such as “FUN1”, “FUN2”, and user-defined variables as “VAR1”, “VAR2”. At this time, we can get a dataset of code fragments.

To obtain the vector representations that suitable for the input to models, we split each fragment into a set of tokens and choose Word2Vector model to vectorize them. Deep learning models exploit different dimension of vectors for training. In this paper, we build six models as smart contract vulnerability detectors. And we will perform further evaluation based on them.

5.2. Implementation Detail

All experiments are conducted on a computer equipped with an Intel(R) Core(TM) i7-4710MQ CPU@2.5 GHz, and 32 GB RAM. We exploit Keras with Tensorflow backend to implement our approach. In the Bi-GRU part of the model, the number of layers is two, the number of hidden layer units is set to 128, and the output dimension of single attention encoder is 256. The number of heads in the multi-head attention encoder is set to 8, and the output dimension \( d = 512 \). The hybrid attention encoder generates feature vectors of 768 dimensions. The optimizer is Adam and all parameters are set to default. The learning rate is initialized to 0.002. To avoid overfitting, dropout is set for every two layers of the network and the ratio is set to 0.1. Cross-entropy loss is used as the loss function, the batch size is set to 64, and the epoch is 100.

In order to reasonably evaluate the test results of the model, we select the evaluation metrics that are widely used in machine learning tasks, including Accuracy, Recall, Precision, and F1-score. Some of the indicators are calculated according to the confusion matrix, which is defined in Table 3.

Table 3. Confusion matrix.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>True Positive(TP)</td>
<td>False Negative(FN)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>False Positive(FP)</td>
<td>True Negative(TN)</td>
<td></td>
</tr>
</tbody>
</table>

Based on the confusion matrix, the specific formulas of the above evaluation indicators are as follows:

\[
\text{Precision} = \frac{TP}{TP + FP} \quad (26)
\]

\[
\text{Recall} = \frac{TP}{TP + FN} \quad (27)
\]
\[ F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]  
(28)

\[
\text{Accuracy} = \frac{TP + TN}{TN + TP + FP + FN} 
\]  
(29)

where \(TP\) means that the actually vulnerable contract code is detected as “vulnerable”; \(FP\) indicates that the actual contract code without vulnerability is detected as “vulnerable”; \(FN\) shows actual vulnerabilities are detected as “non-vulnerable”. Accuracy represents the proportion of correctly predicted data in total samples; Precision shows the proportion of correctly predicted samples in actual correct samples; Recall is the proportion of actual positive samples in predicted positive samples. The F1-score considers both the precision rate and recall rate, which can well reflect the detection effect of the model.

5.3. Experimental Results

Result for answering RQ1: To answer whether our proposed model can be improved by incorporating critical code fragment, we conducted two experiments on the SmartWild dataset. It can be seen that the number of positive (i.e., contain vulnerability) and negative samples is heavily imbalanced. Therefore, we limited the generation of negative examples in extracting code fragments from the source code. For evaluation, we used 1864 samples from SmartWild and all smart contracts from Sbcurated. In total, the number of testing set was 2000. We built two datasets, including the original contracts set and the extracted code fragments set. Table 4 shows that the experimental results with code fragments extraction are better than source code as a model input. Obviously, there is a significant improvement in each of the metrics. In particular, Precision and Recall is improved by 3.2% and 7.08%. These results prove that the extracted code fragments can improve the effectiveness of the model. Conceptually, we infer that the original source code of smart contracts contains some information unrelated to vulnerabilities, and the code fragments with data flow dependency and control flow dependency between statements pay more attention to the key point of vulnerability and learn the relevant vulnerable features better.

Table 4. The effect of code fragment on vulnerability detection.

| Model                  | Number(P|N)   | Accuracy | Precision | Recall | F1     |
|------------------------|-------------|----------|-----------|--------|--------|
| HAM (Unsliced)         | 2000(1183|817) | 84.57%    | 86.38%  | 84.56% | 85.46% |
| HAM (code fragment)    | 2000(1064|936) | 88.96%    | 89.58%  | 91.64% | 90.51% |

Result for answering RQ2: To prove the effectiveness of our proposed method, we focused on the performance comparison from two categories. First, we chose several representative deep learning models based on data, which include RNN, LSTM, GRU, and so on. The comparison results are summarized in Table 5.
Table 5. Performance Comparison of Deep Learning Models on Code Fragments.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Performance Metrics</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>RNN</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>72.28%</td>
<td>86.02%</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>57.40%</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>52.54%</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>54.86%</td>
</tr>
<tr>
<td>Arithmetic Vulnerability</td>
<td>Accuracy</td>
<td>67.19%</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>76.57%</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>54.56%</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>63.72%</td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>Accuracy</td>
<td>72.18%</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>56.64%</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>51.58%</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>53.99%</td>
</tr>
<tr>
<td>Timestamp Dependency</td>
<td>Accuracy</td>
<td>65.76%</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>61.46%</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>54.78%</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>57.97%</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>Accuracy</td>
<td>64.06%</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>73.57%</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>52.66%</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>61.38%</td>
</tr>
</tbody>
</table>

As can be seen from Table 5, it can be obviously concluded that the HAM outperforms other baseline methods by a large margin in terms of all the four metrics. In particular, HAM achieves a high F1-score of 94.04%, an accuracy of 93.36%, and a recall of 96.64%. An improvement may stem from the fact that we consider data flow and control flow in smart contracts. One can see that the Recall rate and F-score of BiLSTM and BiGRU are higher than the corresponding LSTM and GRU. We think the reason is that in the field of smart contract vulnerability detection, the effect of bidirectional recurrent neural network is better than that of unidirectional recurrent neural network. This is because during the training process of the neural network model, the next output of the unidirectional neural network is only affected by the previous output, whereas the output of the bidirectional neural network is not only affected by the forward output, but also by the backward output. The output of the bidirectional neural network is jointly determined by the output of several units before and after. However, the HAM exploits a single-head attention encoder combined with a bidirectional network and a multi-head attention encoder, which can better perform feature learning.

We further visualize the quantitative results of Table 5 in Figure 12a–e. Specially, Figure 12a–e present the comparison results of reentrancy, arithmetic vulnerability, unchecked return value, timestamp dependence and tx.origin vulnerability detection, respectively. Among them, the seven rows from front to back represent model RNN, LSTM, GRU, BiLSTM, BiGRU, BiGRUAtt, and our proposed method HAM. For each row in the Figures, accuracy, precision, recall, and F1 score are respectively presented from left to right. Additionally, considering that only the reentrancy vulnerability can be detected by all three traditional tools, we present only its visual results in Figure 12f, where the four rows from front to back refer to Smartcheck, Oyente, Securify, and HAM, respectively. It is obvious that HAM outperforms existing methods.

In order to further evaluate the detection effect of smart contract vulnerability detection technology based on deep learning, this paper investigates existing smart contract vulnerability detection tools, and selects three of them, such as Oyente, SmartCheck, and Securify, to compare with our proposed solution. The three tools have been widely used in the field of smart contract security. Table 6 shows the performance comparison results,
N/A denotes that the tool does not support the vulnerability. Thus, we have the following observations. From the table, we find that the proposed HAM in this paper outperforms the other three methods with an accuracy of 93.36%, a precision of 91.58%, recall of 96.64%, and an F1-score of 94.04%. However, there is not much difference in the recall rates of the five vulnerabilities, suggesting that HAM pays more attention to the detection rate of smart contracts with vulnerabilities. SmartCheck relies on some rigid and simple logical rules to detect vulnerabilities, which leads to low accuracy and F1-score. In addition, Securify has a highest recall of 92.48% on unchecked return value. In addition, the precision rate of the three traditional methods is low, and they have a high false positive rate in the detection process. The HAM method not only has a higher recall rate but also improves the precision rate, which is more conducive to vulnerability detection.

Result for Answering RQ3: To demonstrate whether our model can detect multiple kinds of smart contract vulnerabilities, the proposed HAM in this paper was trained and tested for five common vulnerabilities, including reentrancy attack, timestamp dependency, arithmetic overflow, tx.origin, and unchecked return value. Arithmetic overflow refers to Integer overflow and Integer Overflow. Table 5 shows the quantitative results, in which we observe that all deep learning models can detect the target vulnerabilities to some extent. Apart from this, except for HAM, the performances of the other three approaches are relatively low, whose F1-score and recall are about 60% and 50%, respectively. Apart from the Arithmetic overflow having an F1-score of 75.57% with the HAM model, the reason is that the features of this vulnerability are not obvious in the Solidity code. As for Reentrancy and Timestamp, they have rich semantic features that are easier to identify. This results in slight accuracy and recall. In summary, the proposed approach is able to automatically learn the vulnerability feature from the vector representation and has good generalization ability.
It is obvious that HAM outperforms existing methods.

Figure 12. Visualization of the quantitative results: (a–e) present comparison results of reentrancy vulnerability, arithmetic vulnerability, unchecked return value, timestamp dependence and tx.origin vulnerability detection, respectively. (f) shows comparison results of reentrancy vulnerability detection on Smartcheck, Oyente, and Securify.
Table 6. Performance comparison with three existing state-of-the-art methods.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Performance Metrics</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Smartcheck</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>64.89%</td>
<td>72.39%</td>
</tr>
<tr>
<td></td>
<td>32.16%</td>
<td>53.47%</td>
</tr>
<tr>
<td></td>
<td>24.79%</td>
<td>47.78%</td>
</tr>
<tr>
<td></td>
<td>28.00%</td>
<td>50.47%</td>
</tr>
<tr>
<td>Arithmetic Vulnerability</td>
<td>42.19%</td>
<td>67.36%</td>
</tr>
<tr>
<td></td>
<td>38.24%</td>
<td>37.13%</td>
</tr>
<tr>
<td></td>
<td>36.50%</td>
<td>27.39%</td>
</tr>
<tr>
<td></td>
<td>37.35%</td>
<td>31.52%</td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>38.66%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>32.50%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>87.54%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>47.40%</td>
<td>N/A</td>
</tr>
<tr>
<td>Timestamp Dependency</td>
<td>58.34%</td>
<td>60.32%</td>
</tr>
<tr>
<td></td>
<td>48.12%</td>
<td>46.68%</td>
</tr>
<tr>
<td></td>
<td>27.36%</td>
<td>30.48%</td>
</tr>
<tr>
<td></td>
<td>34.89%</td>
<td>36.88%</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>70.86%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>49.85%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>54.86%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>52.24%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6. Discussions and Conclusions

The security of smart contracts is of great significance to the sustainable development of blockchain. To enhance the performance of traditional approaches, we propose a deep learning-based hybrid attention mechanism model for smart contract vulnerability detection. This model makes full use of the advantage that hybrid attention can extract more semantic information, which is composed of a single-head attention encoder and a multi-head attention encoder. It operates on four phases, namely, code fragment extraction, word embedding, feature learning, and vulnerability detection. We adopted the Word2Vec technology to complete the word embedding of smart contracts. Then the feature vectors of smart contracts are learned and extracted using two branches of HAM. Finally, the obtained features are concatenated and input to the fully connected layer and softmax activation for classification.

We have proven the performance of HAM with a wide range of experiments. First, we compared the HAM on code fragments with unsliced smart contracts. The improvement for each metric is obvious with an accuracy of 88.96%, precision of 89.8%, recall of 91.64%, and F1 of 90.51%, improved by 4.39%, 3.2%, 7.08% and 5.05%, respectively. Then, we chose six deep learning-based models, such as RNN, GRU, and LSTM, and three popular vulnerability detection tools, such as Smartcheck, Oyente, and Securify, for comparison with our proposed HAM. The results show that HAM model achieved better prediction accuracy of 93.36%, 80.85%, 82.56%, 85.62%, and 82.19% in reentrancy vulnerability, arithmetic vulnerability, unchecked return value, timestamp dependency, and tx.origin, respectively. In addition, the results also indicates that HAM is capable of detecting multiple smart contract vulnerabilities.

We experimentally demonstrate that the HAM model significantly outperforms other nine advanced vulnerability detection methods with higher accuracy again a large number of smart contract vulnerabilities. However, it needs to be noted that existing deep learning-based smart contract vulnerability detection methods are mostly black box detection processes, which only present final vulnerability detection results by training models. Due to the inherent “black box” nature of deep learning model, the specific internal working state and processing process are not transparent, so there is no reasonable interpretation.
about the vulnerability detection results. Hence, the deep learning model should consider how to provide its reasonable explanatory description for unconvincing results. In addition, it is worth mentioning that expert rules defined in traditional detection tools are also powerful tools for analyzing contract vulnerabilities. The future deep learning model should be integrated with expert rules related to vulnerabilities in traditional detection methods, so as to better improve the accuracy of vulnerability detection.

**Author Contributions:** Conceptualization, H.W. and Y.H.; methodology, H.D. and H.W.; software, H.D.; writing—original draft preparation, H.D.; writing—review and editing, H.W.; visualization, Q.D.; supervision, H.W.; project administration, H.W. and Y.H.; funding acquisition, H.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** The work is supported by Major Public Welfare Projects Foundation of Henan Province (20130021020) and Open Foundation of Henna Key Laboratory Cryptography (LNCT2020-A08).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

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

**References**


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.