A Smart Contract Vulnerability Detection Method Based on Multimodal Feature Fusion and Deep Learning

Jinggang Li 1, Gehao Lu 1,*, Yulian Gao 1 and Feng Gao 2

1 School of Information Science, Yunnan University, Kunming 650500, China; ljg2626@mail.ynu.edu.cn (J.L.); gaoyulian@itec.ynu.edu.cn (Y.G.)
2 Yunnan Provincial Health and Medical Big Data Center, Kunming 650500, China; ynsjkyldsjxz2802@163.com
* Correspondence: glu@ynu.edu.cn

Abstract: With the proliferation of blockchain technology in decentralized applications like decentralized finance and supply chain and identity management, smart contracts operating on a blockchain frequently encounter security issues such as reentrancy vulnerabilities, timestamp dependency vulnerabilities, tx.origin vulnerabilities, and integer overflow vulnerabilities. These security concerns pose a significant risk of causing substantial losses to user accounts. Consequently, the detection of vulnerabilities in smart contracts has become a prominent area of research. Existing research exhibited limitations, including low detection accuracy in traditional smart contract vulnerability detection approaches and the tendency of deep learning-based solutions to focus on a single type of vulnerability. To address these constraints, this paper introduces a smart contract vulnerability detection method founded on multimodal feature fusion. This method adopts a multimodal perspective to extract three modal features from the lifecycle of smart contracts, leveraging both static and dynamic features comprehensively. Through deep learning models like Graph Convolutional Networks (GCNs) and bidirectional Long Short-Term Memory networks (bi-LSTMs), effective detection of vulnerabilities in smart contracts is achieved. Experimental results demonstrate that the proposed method attains detection accuracies of 85.73% for reentrancy vulnerabilities, 85.41% for timestamp dependency vulnerabilities, 83.58% for tx.origin vulnerabilities, and 90.96% for integer overflow vulnerabilities. Furthermore, ablation experiments confirm the efficacy of the newly introduced modal features, highlighting the significance of fusing dynamic and static features in enhancing detection accuracy.

Keywords: vulnerability detection; smart contract; multimodal fusion; deep learning

MSC: 68T07

1. Introduction

Blockchain plays a crucial role in various decentralized technologies and applications, including distributed storage, computing, security, interaction, and transactions [1]. Since 2008, with the increasing popularity of cryptocurrencies such as Bitcoin and Ethereum [2] in the financial markets, related blockchain technology has rapidly developed and matured, becoming one of the most promising network information technologies for ensuring security and privacy [3]. Technically, distinct from traditional security solutions, blockchain is fundamentally a decentralized shared ledger, with its core objectives being the safeguarding of data security, providing immutability, and ensuring trustworthiness.

As shown in Figure 1, smart contracts are protocols stored in the blockchain in the form of program code, which automatically execute when specific predefined conditions are met. They are essentially designed to automatically execute protocols between untrusted parties without the need for a trusted third party to intervene. Ethereum provides a decentralized ecosystem that utilizes a programming language called Solidity and a Turing-complete machine called the Ethereum Virtual Machine (EVM) to support smart contracts [4].
general process of a smart contract is as follows: firstly, protocol terms are written in a high-level language (such as Solidity) to create the source code. Next, this is compiled into bytecode (for example, EVM code). Subsequently, the bytecode is uploaded to the blockchain in the form of a transaction and stored in a block of the distributed ledger. Once the predefined conditions are met, these bytecode instructions or opcodes are translated into operations that run on the EVM, thereby executing the corresponding business logic [5].

![Smart Contract Diagram](image)

**Figure 1.** Smart contract execution process.

However, the widespread adoption of smart contracts has also brought about a series of security challenges. The immutability feature of blockchain, once a vulnerability exists in the code of a smart contract, can potentially lead to significant losses for the associated account [6]. In 2016, an organization named “The DAO” fell victim to a hacker attack, resulting in the theft of millions of ether due to a vulnerability in its smart contract code [7]. In July 2021, multiple contract projects, including Chainswap and Anyswap, were successively attacked, resulting in a cumulative loss exceeding $7.87 million [8]. In October 2022, BNB Chain, one of the most active public chains globally, experienced a security breach. The hacker exploited a cross-chain bridge vulnerability to acquire 2 million BNB in two separate attacks, with an estimated value of approximately $566 million. According to statistics from the SlowMist website [9], as of 2023, cumulative losses from attacks on Ethereum smart contracts have exceeded $3.1 billion. These recurrent incidents of smart contract security have significantly hindered the development of smart contracts, as well as the subsequent progress of Ethereum and other blockchain companies. Therefore, it is imperative to conduct comprehensive and accurate security analyses on smart contracts before their deployment on the Ethereum platform, aiming to eliminate vulnerabilities to the greatest extent possible and prevent the occurrence of security issues.

Smart contracts face potential threats from various vulnerabilities, including reentrancy vulnerabilities, integer overflows, timestamp dependencies, and others. These vulnerabilities can cause unforeseen contract behavior during runtime, diverse attack types, and potentially severe outcomes like data leakage and financial losses. Comprehensive vulnerability detection methods are essential to encompass multiple smart contract vulnerabilities. By addressing a spectrum of vulnerabilities, these detection techniques can identify and resolve various security concerns, enhancing smart contract security and stability, thereby minimizing the risks associated with financial losses and data breaches. Simultaneously, these methods contribute to constructing a more secure and dependable blockchain ecosystem, fostering the continual evolution of blockchain technology and its applications.
The causes of smart contract security vulnerabilities can be viewed as functions running out of control. Attackers may exploit problematic code segments to perform additional operations, such as altering the function’s timestamp, recursive functions, etc., to introduce security risks. The security assessment of smart contract vulnerabilities has become a hot research topic in the field of blockchain security, gradually forming research directions including static analysis, dynamic analysis, and combined static–dynamic analysis [10]. At the same time, these studies face some challenges: static analysis tends to have a high false-positive rate and relies excessively on detection rules; dynamic analysis has a high false-negative rate due to path coverage issues, and it is difficult to address issues related to high time and space costs for a large number of contracts; combined static–dynamic methods generally guide fuzzing or symbolic execution based on the results of static analysis, depending on the specific vulnerability type. This approach can improve analysis efficiency while ensuring accuracy, but there are still significant limitations to its capabilities [11].

Given that potential vulnerabilities in smart contracts can lead to serious consequences, and the current means of detecting vulnerabilities are relatively lacking in effectiveness, this paper proposes a novel method for detecting vulnerabilities in smart contracts. This method combines dynamic execution and static scanning to extract both dynamic and static vulnerability features from the contracts. Finally, a neural network is employed to train a contract classification model for detecting vulnerabilities in smart contracts. To validate the effectiveness of the proposed method, extensive experiments were conducted on a dataset of smart contracts, and comparisons were made with competitive vulnerability detection tools. The main contributions of this paper are as follows:

1. By combining various word-embedding methods with deep learning techniques, this approach extracts comprehensive and high-coverage static features from Solidity source code (SC) and Ethereum Virtual Machine (EVM) bytecode.

2. Executing contracts and utilizing instrumentation to gather critical stack information during execution enable the extraction of dynamic vulnerability features. The fusion of static and dynamic features is applied to smart contract vulnerability detection, facilitating the identification of various types of smart contract vulnerabilities while maintaining excellent detection performance.

3. Experimental results demonstrate that the proposed method, which integrates the strengths of deep learning and hybrid learning, surpasses single-modal features. The incorporation of multi-modal and multi-type features leads to superior performance in detecting smart contract vulnerabilities.

2. Related Work

2.1. Traditional Methods

There has been extensive research in the field of vulnerability detection in smart contracts. Based on the software testing techniques employed to achieve their goals, existing methods can be categorized into four types: fuzz testing, symbolic execution, formal verification, and static analysis. Oyente [12], proposed by Luu et al., is a smart contract vulnerability detection tool based on symbolic execution. It is the first technology for detecting vulnerabilities in smart contracts and the first to utilize symbolic execution for this purpose. SmartCheck, introduced by Tikhomirov et al., is an extensible static analysis tool. It converts Solidity source code into an XML-based intermediate representation and then performs checks using XPath patterns [13]. Mythril [14] is the official smart contract vulnerability detection tool for Ethereum, capable of detecting security issues in a large number of smart contracts. The main idea behind it is to use symbolic execution to explore all possible unsafe paths. Security [6] introduces a formal verification-based vulnerability detection method, using a specified domain-specific language to describe all compliance and violation patterns for detecting smart contract EVM bytecode. Slither [15] is a static analysis tool that identifies potential security vulnerabilities by examining the contract’s source code. Contracctfuzzer [16] uses fuzz testing for vulnerability detection, leveraging the EVM to record the runtime state of smart contracts, successfully detecting vulnerabilities
leading to DAO contract events, as well as vulnerabilities in the Parity wallet. sFuzz [17], based on AFL, employs feedback-adaptive fuzzing strategies to enhance test-case coverage, achieving vulnerability detection. ConFuzzius2 [18] combines symbolic execution and fuzz testing using a hybrid fuzz testing approach to improve the detection of deep-seated errors. These tools primarily rely on expert-predefined hard logic rules for vulnerability detection. However, with the advancement of smart contract technology, these tools are no longer able to meet current demands. Traditional smart contract vulnerability detection tools have certain limitations, which are outlined as follows:

1. The structure and functionality of smart contracts are becoming increasingly complex. With the continuous increase in the types of vulnerabilities in smart contracts, rules defined by experts based on vulnerability definitions are unable to keep up with the pace of updates to smart contract vulnerabilities. The overlay of multiple coarse logical rules defined by experts may lead to a high false-positive rate, thereby increasing the cost of manually filtering vulnerabilities.

2. Smart contracts may exhibit different behaviors at runtime based on varying conditions. This makes formal verification challenging in handling dynamism and conditionality. Symbolic execution may lead to a proliferation of execution paths, particularly for complex smart contracts, making the analysis highly time- and resource-intensive. Fuzz testing is a method with a high degree of randomness, generating random input data, thus it may not guarantee the detection of all possible vulnerabilities. Both fuzz testing and symbolic execution require specific or symbolic execution of smart contracts, necessitating a sandbox environment and incurring execution overhead.

2.2. Deep Learning Methods

In recent years, deep learning technology has been applied in various fields, including the field of vulnerability detection. Among them, methods based on text processing have been widely utilized. Tann WJW and others converted the opcodes in the opcode sequence into one-hot vector representations, and then transformed them into lower-dimensional vectors through embedding algorithms, while capturing the sequential relationships between opcodes [19]. Experimental results have shown that sequence learning methods outperform analysis tools based on symbolic execution. Huang J and colleagues proposed a smart contract vulnerability detection model based on multi-task learning. This model consists of shared layers and specific task layers and constructs a shared-layer network based on an attention mechanism to learn the key features of smart contract opcode sequences [20]. CBGRU [21] is a hybrid deep learning model that uses Word2Vec [22] and FastText [23] for word embedding. These serve as two input branches for the feature extraction layer, and finally, in the classification layer, the features extracted from both branches are fused through a connection layer. The aforementioned text-based methods only focus on the sequence relationships in the code and consider only a single modality of data. They overlook the most critical structural information, which prevents them from comprehensively learning the structural features of the code.

Yu X and his team developed a comprehensive smart contract vulnerability framework called DeeSCVHunter, introducing the concept of Vulnerability Candidate Slices (VCS) [24]. Wu H and colleagues adopted a smart contract representation method based on critical data flow graph information, capturing the essential features of contracts before conducting vulnerability detection. They proposed a new tool called Peculiar [25]. Reference [26] first introduced a smart contract vulnerability detection method based on Graph Neural Networks. In the training phase of the Graph Neural Network, a novel message propagation mechanism called Temporal Message Propagation (TMP) was proposed, which updates the information propagation along the edges in chronological order. Reference [27] put forward a Degree-Free Graph Convolutional Neural Network (DR-GCN) and a Temporal Message Propagation Network (TMP) for vulnerability detection, learning from the normalized graphs. The aforementioned smart contract vulnerability detection methods combine static
analysis with deep learning. They use static analysis methods to extract structural information from smart contract code, including Control Flow Graphs (CFGs), Abstract Syntax Trees (ASTs), and data flow graphs (DFGs). Through deep learning methods, especially Graph Neural Networks, they further map structural features to vector spaces, thereby enhancing the effectiveness of smart contract vulnerability detection.

Current methods for extracting features from smart contracts primarily operate on a single level, be it source code, bytecode, or opcodes. Source code provides a high-level abstracted logical structure but lacks information about the actual execution happening at a lower level. Opcodes, on the other hand, depict the actual execution process of the smart contract on the virtual machine, encompassing more detailed execution information. For instance, vulnerabilities like integer overflows may arise at the bytecode level, and this information can only be unearthed through opcode analysis. Opcodes serve as a low-level representation of smart contracts, accurately mirroring the contract’s internal execution process. Generally, they do not encompass high-level semantic information. This implies that models may overlook advanced logic and structural elements present in the source code, making it challenging to comprehend the smart contract’s intent. Furthermore, most existing models for smart contract vulnerability detection primarily rely on static features for learning and detection. However, features obtained through static analysis are limited, as most smart contract vulnerabilities occur during runtime.

2.3. Feature Fusion

As shown in Figure 2, feature fusion [28] refers to the integration of features obtained from different data sources or different feature extraction methods to obtain a more comprehensive and informative representation of features. In the fields of machine learning and deep learning, feature fusion is commonly used to enhance model performance, improve the model’s ability to represent complex data, and address situations involving multi-modal data. In the context of smart contract vulnerability detection, feature fusion involves integrating features from different stages of the smart contract lifecycle and different natures of features. This integration can be achieved through the comprehensive utilization of multiple modalities, including static and dynamic features.

Static feature extraction often relies on deep learning methodologies, particularly employing natural language processing techniques like Word2Vec, GloVe, and FastText on contract source code or bytecode (opcode) to generate word embeddings. Alternatively, graph-based approaches such as GCN, GNN, and Node2Vec are utilized to obtain feature vectors through graph embedding. Dynamic feature extraction involves analyzing smart contract execution paths under varying inputs or conditions. This encompasses capturing details like code paths traversed during runtime, sequences of function calls, conditional branches, loops, and other relevant information. Alternatively, smart contract simulators or virtual machines can be employed for simulated executions, allowing code instrumentation, runtime data recording, and subsequent extraction of dynamic features.

---

![Figure 2. Feature fusion process.](image_url)
Typical feature fusion methods include simple concatenation, weighted summation, multiplication, and more complex model structures that organically combine information from different modalities. The goal of feature fusion is to provide more comprehensive information to better describe the complexity and diversity of data, thereby enhancing the model’s understanding and generalization capabilities. In the context of smart contract vulnerability detection, feature fusion contributes to more accurate vulnerability detection and improved model performance by considering both static and dynamic features of smart contracts.

After feature fusion, it is necessary to update the parameters of the vulnerability detection model through training. This enables the model to learn data patterns based on the fused features and apply these patterns to new, unseen data. The training process helps the model adapt to specific tasks and enhance its performance, ensuring better comprehension of data and accurate predictions based on the fused features. At this stage, models can employ structures such as CNN, RNN, GRU, and bi-LSTM.

Hence, this paper introduces a methodology that involves extracting features from smart contracts, leveraging both dynamic and static characteristics. By merging the strengths of these two feature types and integrating diverse modal features, the goal is to comprehensively capture semantic and inherent logical information, interconnected within the context. This endeavor will aid the model in acquiring a more robust and effective feature representation for detecting smart contract vulnerabilities.

3. Research Method

3.1. Framework

In the assessment of smart contract vulnerabilities, we differ from previous works by not only focusing on a single modality of data source. Instead, we encompass three modalities—source code, opcodes, and instrumentation information—as well as two types of data sources. By selecting state-of-the-art machine learning and deep learning models, we have generated a hierarchical structure of multi-layered features, resulting in a binary decision indicating whether a contract is susceptible to attacks.

In the following lines, we emphasize two critical components of our framework. The first part delves into how to extract and integrate features to obtain a unified multimodal representation. The second part introduces how to feed these features into the artificial intelligence model to detect vulnerabilities in smart contracts. The overall structure of the method is shown in Figure 3.

![Figure 3. Feature fusion vulnerability detection scheme.](image-url)

The source code takes its initial form in the smart contract, and its AST provides a visual representation of the code’s syntactic structure. The SBT, acquired through a comprehensive traversal of the AST, is structured as a sequence. The CFG constructed from opcodes illustrates how each code block is invoked during contract execution, with nodes (N) representing basic opcode blocks. Truffle, a widely used Ethereum smart contract development framework, is employed to capture and record crucial operational steps of the EVM during contract execution.
3.2. Extracting Static Features of Smart Contracts

As shown in Figure 4, in this section we will demonstrate how to conduct data preprocessing and obtain features from the SC layer and EVMO layer. We will further elaborate on how to combine these features for smart contract vulnerability detection.

![Figure 4](image1.png)

**Figure 4.** Overview of the static feature extraction process. It includes two modalities and their respective features. The SC layer extracts features from the contract source code. The EVMO layer extracts relevant features from the contract bytecode.

3.2.1. Source Code Layer

Smart contract source code is inherently unstructured, requiring the extraction of structural features for improved contract detection [29,30]. Hu et al. [31] introduced the Structure-Based Traversal (SBT) method, which involves transforming the AST into a specialized format through a comprehensive traversal. In the SBT representation, “type” signifies structural information, “value” represents lexical information, and brackets indicate the hierarchical structure. Previous studies [32–34] have illustrated the effectiveness of SBT in preserving both the code’s structural and lexical aspects. Consequently, we treat the SBT sequence as a modality of the AST, employing the SBT method to encapsulate the overall semantic information, including both structural and lexical elements.

Using Figure 5 as an illustration, our initial step involves employing the open-source parsing tool, tree-sitter-solidity (https://github.com/JoranHonig/tree-sitter-solidity, accessed on 12 September 2023), to parse the source code and generate an AST. Subsequently, we utilize the SBT method to transform this AST into an SBT sequence. In the provided figure, consider the StockExchange contract that defines a function named “withdraw”. Non-leaf nodes are denoted by type (e.g., the root node of the contract is FunctionDefinition), and various elements such as variables, function names—return value names are represented by “#”. Leaf nodes correspond to values associated with each type.

![Figure 5](image2.png)

**Figure 5.** The SBT process.
For the initialization of the word-embedding matrix, two primary approaches are considered: random values or pretrained vectors obtained from models like CodeBERT [35], Word2Vec [22], GloVe [36], FastText [37], ELMo [38], etc. While many methods are originally designed for natural language text, audio, video sequences, or knowledge graphs, CodeBERT stands out as a pretrained model uniquely suited for handling bimodal data, accommodating both natural language text and programming languages.

In the context of SBT, which incorporates distinctive tokens (e.g., ModifierDefinition, ModifierInvocation, SimpleName, FunctionDefinition), the construction of the SBT corpus requires the inclusion of these additional tokens. Subsequently, the SBT word-embedding matrix is formed based on this corpus and in conjunction with CodeBERT.

3.2.2. EVM Opcode Layer

To enhance the representation of the binary code in smart contracts, a CFG is utilized, encompassing both data dependency and control relationships. The construction of the CFG entails a sequential process: initially compiling the source code of the contract and subsequently decompiling the bytecode to produce EVM opcode instructions along with their corresponding parameters. The illustrated process of compiling source code into opcodes is presented in Figure 6.

![Figure 6](image.png)

**Figure 6.** The process of compiling source code into opcodes.

The CFG is structured with basic blocks, each comprising instructions and their respective parameters. Basic blocks initiate with non-branching instructions and conclude with branching or terminating instructions (e.g., STOP, JUMP, JUMPI, RETURN, REVERT, SELF-DESTROY, etc.). The process involves decompiling the contract bytecode to form basic blocks. The interconnection of these blocks is established by parsing branching instructions (JUMP and JUMPI) within each block, forming the control flow graph of the targeted contract. Nodes within the contract and the edges between them constitute the CFG. The CFG representation is further characterized by two types of basic jumps: unconditional jumps, occurring when a JUMP opcode immediately follows a PUSH opcode, and conditional jumps, arising when a JUMPI opcode immediately follows a PUSH opcode. The resulting CFG is illustrated in Figure 7.

Because nodes are closely related to each other in time, rather than isolated, in order to capture the semantic dependencies between nodes, we constructed three types of edges: conditional jump edges, unconditional jump edges, and sequential flow edges. Each node represents a basic block, and we used the set \( V \) to represent all nodes in the CFG. The call relationships between nodes are considered as edges, and we used the set \( E \) to represent all edges in the CFG.

Nodes within the CFG exhibit close temporal interrelations rather than isolation. To capture the semantic dependencies between nodes, three types of edges are constructed: conditional jump edges, unconditional jump edges, and sequential flow edges. Each node corresponds to a basic block, denoted collectively as the set \( V \), representing all nodes in the CFG. The call relationships among nodes are portrayed as edges, forming the set \( E \), which encapsulates all edges in the CFG.
Following node partitioning and edge construction, achieving standardization of opcodes is essential for expressing the semantic information of the smart contract in the bytecode control flow graph. Uniform labels were assigned to the data following different opcodes to ensure consistency. The specific label for each dataset depends on the corresponding opcode. In alignment with the research by Zhang et al. [5], we categorized the opcodes into six types based on the operand’s nature: arithmetic, block, logic, memory, storage, and bitwise. Refer to Table 1 for the detailed classification.

Table 1. Opcodes and their corresponding data labels.

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Data Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, MUL, SUB, EXP, SIGNEXTEND</td>
<td>ArithData</td>
</tr>
<tr>
<td>BLOCKHASH, COINBASE, TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>NUMBER, DIFFICULTY, GASLIMIT</td>
<td>BlockData</td>
</tr>
<tr>
<td>LT, GT, SLT, SIGE, ISZER0</td>
<td>LogicData</td>
</tr>
<tr>
<td>MLOAD</td>
<td>MemData</td>
</tr>
<tr>
<td>SLOAD</td>
<td>StoreData</td>
</tr>
<tr>
<td>BYTE, SHL, SHR, SAR, AND, OR, XOR, NOT</td>
<td>BitData</td>
</tr>
</tbody>
</table>

Certain opcodes are succeeded by numbers indicating the byte length of the current operation data. Despite these length specifications, the precise length of the operation data corresponding to the opcode does not impact the semantic analysis of the bytecode. Therefore, it is crucial to simplify opcodes with operand lengths of this kind to maintain semantic consistency. Additionally, opcodes like DUP, POP, and SWAP, which directly manipulate stack data in the Ethereum Virtual Machine, have no practical impact on the semantics of smart contracts. Consequently, we opted to remove them. Refer to Table 2 for examples of opcodes after normalization.
Table 2. Standardized opcodes.

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Normalized Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG0-LOG4</td>
<td>LOGX</td>
</tr>
<tr>
<td>PUSH1-PUSH32</td>
<td>PUSHX</td>
</tr>
<tr>
<td>DUP1-DUP16</td>
<td>-</td>
</tr>
<tr>
<td>SWAP1-SWAP16</td>
<td>-</td>
</tr>
<tr>
<td>POP</td>
<td>-</td>
</tr>
</tbody>
</table>

Various graph-based embedding methods, such as DeepWalk [39], Node2Vec [40], and GCNs (Graph Convolutional Networks) [41], have been developed. In relevant studies, experimental results have indicated that GCNs, when applied to process smart contract graphs, adeptly integrate information from neighboring nodes. This integration allows node representations to comprehensively encompass the entire graph structure, rather than relying solely on local information from adjacent nodes. Moreover, a GCN has the capability to preserve the relative positions of nodes in the graph, maintaining the topological structure between nodes. This preservation of topological relationships is crucial for smart contract vulnerability detection, given that vulnerabilities are often closely tied to how nodes are interconnected. Consequently, we opt to utilize a GCN for embedding in the EVM opcode layer.

In the context of graph representation, if we denote the CFG as G, it can be expressed as $G = (V, E)$, where V represents nodes and E represents edges. By training the CFG using a GCN, we can derive embedded features for the graph. Each layer of the neural network can be represented by the following non-linear function:

$$H^{(l+1)} = f(H^{(l)}, A), \quad 0 \leq l < L$$  \hspace{1cm} (1)

where L is the number of layers in the network; the input layer is $H^{(0)} = X$, where X is the feature matrix; the output layer is $H^{(L)} = Z$, where Z is the node-level output; $f(\cdot)$ represents the differentiable function of the neural network; A is the adjacency matrix of G. A GCN for graph classification utilizes both the intrinsic features of nodes and the structural information of the graph. The learning strategy is as follows:

$$\Gamma = \Gamma_0 + \lambda \Gamma_{reg}$$  \hspace{1cm} (2)

$$\Gamma_{reg} = f(X)^T \Delta f(X)$$  \hspace{1cm} (3)

$\Gamma_0$ represents the supervised loss for nodes with labels in the graph and $\Gamma_{reg}$ is the loss introduced by the graph structural information. $\lambda$ is a weight coefficient, and the ideal value for it is $10^{-3}$. X is the node feature matrix and $\Delta$ denotes the Laplacian operator of the graph.

Through the embedding process of the Graph Convolutional Neural Network, we can map the control flow graph of the smart contract to a low-dimensional vector space, enabling subsequent analysis and task processing in this vector space.

3.3. Extracting Dynamic Features of Smart Contracts

Dynamic detection of contract vulnerabilities involves the execution of the contract. In this paper, a dynamic detection pattern for vulnerabilities is designed. During runtime, the system examines the currently executing opcode, determines its type, detects the values of crucial variables in the stack based on this type, and then matches them with the designed dynamic pattern and critical variables. This process is conducted to assess whether the current smart contract exhibits a specific type of characteristic.

- Reentrancy vulnerability

Transferring ether to an Ethereum contract account triggers the execution of the contract account’s fallback function. When sending Ether, the contract executes its cor-
responding fallback function. If a transfer operation is initiated to the address of an attacker’s contract, forcing the execution of the attacker contract’s fallback function, and if the fallback function contains a callback to the attacker’s own code, it leads to the execution of the contract “re-entering” itself. This type of contract vulnerability is known as a reentrancy vulnerability.

The distinctive characteristic of a reentrancy vulnerability is the sequence of transferring ether followed by the update of the contract state. To identify this vulnerability, we initially examine whether the currently executed opcode is CALL. Verification involves checking if the memory contains the address parameter of this opcode and if the position containing the amount parameter is not empty. A positive result indicates that the CALL opcode has indeed transferred ether to an external address, and we duly recorded this operation. Subsequently, we ascertained if the currently executed opcode is SSTORE. The presence of the SSTORE opcode signals the conclusion of the transfer operation and the impending update of the contract state, which prompted us to record this operation as well. Finally, we analyzed the sequence of CALL opcode occurrences in comparison to that of SSTORE occurrences. This sequence delineates the order of the transfer and contract state update. If the CALL opcode precedes the SSTORE opcode, it implies that the contract may be vulnerable to reentrancy attacks.

• Timestamp dependence vulnerability

When smart contracts use block timestamps as triggering conditions for executing certain critical operations, a timestamp dependency vulnerability may arise. For instance, if the timestamp of a future block is used as a seed to generate a random number determining the winner of a game, a miner can exploit the vulnerability by freely setting the timestamp of future blocks within a short timeframe to manipulate the game’s outcome and gain illicit benefits. Attackers first analyze the target contract, identifying dependencies on timestamps within it. They then look for conditions in the contract that can be manipulated, such as controlling certain inputs or executing specific operations. These conditions may impact the logic related to timestamps. Once manipulable conditions are found, attackers execute malicious logic, potentially leveraging timestamps to carry out the attack.

The design concept of the dynamic pattern for timestamp dependency vulnerability is based on a scenario where the current contract uses a timestamp and transfers non-zero ether during execution. Firstly, it can be checked if operations related to the timestamp, such as TIMESTAMP, NUMBER, DIFFICULTY, etc., are called during execution. If these operations are not found, the contract will not be susceptible to timestamp dependency attacks. Next, it can be checked whether ether is sent via ‘send’, and verified if the contract has transactions with external addresses. If there is only a timestamp dependency without any loss of assets, the contract is considered safe. Finally, the operand at the corresponding position when executing the CALL opcode can be examined to ensure that the amount of ether transferred is not zero.

• Tx.Origin vulnerability

The tx.origin vulnerability is a type of smart contract security issue involving the misuse of the tx.origin variable. Attackers may exploit the tx.origin to carry out an attack, where tx.origin is a variable provided in Ethereum contracts to obtain the initiator of the transaction (original sender). After identifying the use of tx.origin in the target contract, attackers create a malicious contract and initiate a transaction through other legitimate contracts or external accounts while calling the target contract. In this transaction, the attacker sets their own address as that of a legitimate user. As the target contract uses tx.origin to verify the caller’s identity instead of msg.sender, the attacker successfully passes the verification and executes malicious logic.

One characteristic of the tx.origin vulnerability is the use of the tx.origin function to retrieve the calling information at the bottom of the stack. The detection pattern for tx.origin can be designed to check if the currently executed opcode is ORIGIN. If this opcode is not
present during the contract execution process, then the contract will not be susceptible to tx.origin vulnerability attacks.

- Integer overflow vulnerability

  The smart contract integer overflow vulnerability is a security issue involving potential overflow situations during integer operations in a contract. Integer overflow occurs when the integer type (e.g., uint) in the contract exceeds the range defined for it during an operation. Attackers may exploit this overflow phenomenon to cause unexpected results, such as incorrect fund transfers or state modifications. Attackers achieve this by sending specific inputs or executing particular operations that cause the integer value in the contract to exceed its defined range, for instance, by using a sufficiently large value in an addition operation to cause overflow. Due to integer overflow, the computed result may become negative or wrap around to a smaller positive number. This can lead to failed conditional checks or erroneous fund transfers in the contract, granting the attacker undue privileges or assets.

  The detection pattern for integer overflow vulnerabilities can be designed to check if the currently executed opcode is ADD, SUB, MUL, or DIV. Then, it can monitor the operands in the stack before the execution of the opcode. For each operand, it checks if its range could potentially lead to an integer overflow. For instance, if the operand is a 32-bit integer, it verifies if it falls within an acceptable range. Finally, it performs the actual check for integer overflow by comparing the value of the operand with the maximum and minimum values of its data type. If the operand exceeds the valid range, an overflow may occur.

  Truffle is a popular Ethereum smart contract development framework. As shown in Figure 8, We extracted its dynamic features by executing smart contracts within this framework. Executing a contract requires providing a set of inputs that conform to the specified parameter types and quantity. Based on the parameter types and quantity, input data is generated. Combined with the dynamic patterns designed above, we can monitor the specific type of the currently executed opcode and selectively obtain the operand at the corresponding position in the stack. Extracting dynamic features using dynamic patterns requires instrumenting the virtual machine at runtime. This is achieved by obtaining the value of key variables or the value at a specific position in the stack. The specific process of extracting dynamic features from smart contracts is shown in Algorithm 1.

![Figure 8](source.png)

**Figure 8.** Running Truffle to obtain the values of critical variables and matching them with the dynamic patterns to obtain dynamic features.
Algorithm 1: Dynamic feature extraction for smart contracts

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog: A Smart Contract Opcodes:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>output: A Smart Contract Opcodes:</td>
</tr>
</tbody>
</table>

Begin
Let output be a list
if prog.has(CALL) && prog.has(SSTORE) && indexof(CALL) && indexof(SSTORE) then
update output
end if
if prog.has(Timestamp opcodes) && prog.CALL.hasDestination() && prog.CALL.ether !=0 then
update output
end if
if prog.has(Integer overflow) then
update output
end if
if prog.has(ORIGIN) then
update output
end if
return output
end

For patterns closely associated with specific vulnerabilities, we utilized one-hot vectors to represent each pattern. The vulnerability detection patterns designed in this paper encompass four types. Consequently, we employ four binary flags for encoding, appending a binary digit (0/1) to each vector to indicate whether the tested function contains that pattern. In this approach, the one-hot vector for each smart contract signifies its dynamic features.

3.4. Vulnerability Detection Model Design

This section establishes the framework that connects the hierarchical structure of fused features with state-of-the-art artificial intelligence models in a multimodal setting. Figure 9 presents an overview of these tasks. In our framework, each task denotes a vulnerability detection branch, encompassing pipeline input feature selection, feature dimensionality unification, feature fusion, model training, and decision-making.

![Figure 9](https://via.placeholder.com/150)

**Figure 9.** Smart contract vulnerability detection based on multi-modal feature fusion architecture. The features are input into the Multi-Modal Feature Fusion Network. The network consists of four stages: (1) standardizing feature dimensions, (2) feature fusion, (3) model training, (4) classification decision.

1. Feature dimension unification: When combining different features, at least one dimension should be equal. The Spatial Pyramid Pooling (SPP) layer can generate fixed-size outputs, reducing the likelihood of overfitting while retaining detailed information
Taking this into consideration, we employ the SPP layer to unify the dimensions of static and dynamic features.

2. Feature fusion: After the dimension unification stage, we proceed with feature fusion. In this paper, we choose to perform feature fusion through horizontal concatenation (Concat). Concatenation is an operation that combines two or more tensors along a certain dimension. When performing the Concat operation, it is essential to ensure that the dimensions being concatenated are of the same size.

3. Model training: Following the feature fusion stage, in the fusion model training phase, we utilize a bi-LSTM model with a self-attention mechanism. In this stage, the bi-LSTM model with self-attention is selected as the fixed strategy selection model. The attention layer takes the output $X$ of the bi-LSTM layer as input for the attention mechanism, calculating attention weights as shown in Formula (4). The attention scores are then normalized using the softmax function, as shown in Formula (5). Finally, the weighted combination of attention weights and the output of the bi-LSTM layer yields the final feature vector output, as illustrated in Formula (6).

$$H = \tanh(X)$$  \hspace{1cm} (4)

$$\alpha = \text{softmax}(w^T \cdot H)$$  \hspace{1cm} (5)

$$V = X \cdot \alpha^T$$  \hspace{1cm} (6)

4. Classification decision: The final stage is decision-making. To train and evaluate the vulnerability detection model, this paper utilizes a random forest (RF) classifier for decision-making. A RF is constructed using the training set, comprising multiple decision trees. Each tree is built based on different random samples and subsets of features. The features obtained from the deep learning model are used as the features for each sample. During the training process of each decision tree, the random forest extracts a portion of samples from random samples and randomly selects a subset of features from all features. This ensures diversity among the trees. The splitting process at each node selects the best feature based on a certain impurity measure (e.g., the Gini coefficient). For each test sample, its feature vector is input into each decision tree, and the final classification result is determined by a vote based on the classification results of each tree, with the class receiving the most votes being the ultimate classification result.

4. Experiment

In this section, we conduct an empirical evaluation of our proposed method using publicly available datasets. To assess the performance of the proposed approach, we formulate the following research questions:

RQ1: Can the proposed method effectively detect common vulnerabilities in smart contracts, and is its vulnerability detection superior to existing methods?

RQ2: Does the proposed method have advantages compared to using a single modality?

RQ3: How does the selection of fusion model affect the training effectiveness?

4.1. Experimental Design

For empirical analysis, this paper utilized the open-source tool tree-sitter-solidity to parse Ethereum Solidity source code into the AST format. Further data optimization operations such as word segmentation and removal of irrelevant terms were performed. A CFG generator for EVMO was developed. In terms of implementing AI models, the TensorFlow, Keras, Gensim, Stellargraph, and PyTorch libraries were used. All experiments were conducted on physical machines running Ubuntu 20.04 operating system, equipped with an Intel Core i7 CPU processor clocked at 3.7 GHz, 32 GB of RAM, and a GeForce RTX 4080 graphics card with 16 GB of video memory. The models were accelerated using
the CUDA 12.0 computing library for training. The experimental code was written in Python 3.10.

Since our approach is based on source code and bytecode, we evaluated them separately. For comparative purposes, we selected state-of-the-art source code-based vulnerability detection tools, namely SmartCheck [13], Mythril [14], Slither [15], and CBGRU [21]. All of them fall under the category of static analysis tools, suitable for source code scenarios. Both Slither and SmartCheck convert Solidity code into an XML-based intermediate representation and detect vulnerabilities by inspecting XPath patterns. CBGRU employs two-word embeddings in the SC mode as inputs for feature extraction at the feature extraction layer. Finally, at the classification layer, features extracted from both branches are fused through a concatenation layer.

Additionally, we also selected three vulnerability detection tools that support opcodes: ContractFuzzer [16], Oyente [12], and Securify [6]. They belong to three different types: fuzz testing, symbolic execution, and formal verification, respectively. We conducted multiple training runs and collected average results. By comparing with the aforementioned methods, we validated the effectiveness and superiority of our approach.

4.2. Dataset

To evaluate the effectiveness of our proposed model, we selected labeled smart contracts from the open-source datasets Smartbug [43] and SolidiFI [44]. Smartbug contains two parts, one part contains 69 smart contracts and 115 marked vulnerabilities, and the other part contains 47,398 unique smart contracts of Ethereum, and the vulnerabilities were marked using nine static analysis tools. Smartbug is one of the most widely used public smart contract vulnerability datasets in the field of deep learning. SolidiFI is a real Ethereum smart contract data set with vulnerability labels, used to supplement Smartbug. The distribution of various types and quantities of vulnerabilities in the prepared dataset is presented in Table 3. We randomly divided the dataset into three subsets: 70% for training, 20% for validation, and the remaining 10% for testing. The training set is utilized to enable the neural network model to learn different types of vulnerabilities in smart contracts. The validation set is employed to fine-tune the model’s parameters during training, preventing overfitting. Lastly, the testing set is used to assess the model’s generalization ability in vulnerability detection. This three-way split of the dataset ensures a comprehensive and robust evaluation of the model.

Table 3. Types and quantities of datasets.

<table>
<thead>
<tr>
<th>Reentrancy</th>
<th>Timestamp Dependency</th>
<th>Tx.Origin</th>
<th>Integer Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1531</td>
<td>1671</td>
<td>1323</td>
<td>1417</td>
</tr>
</tbody>
</table>

4.3. Experimental Setup

The input for GCN includes the dimensionality of each node in the graph (set to 128), the adjacency matrix representation corresponding to the nodes in the graph, and the labels corresponding to the nodes in the graph. The output is the number of graph categories. The batch size is set to 128. The number of layers in the GCN network is set to 3. The weight coefficient $\lambda$ in the GCN learning strategy is set to 0.001.

The activation function for the neural network is the rectified linear unit (ReLU) which is a non-linear function and is less prone to the vanishing gradient problem. The Adam optimizer, commonly used in deep learning, is employed to adjust the network parameters. The learning rate for GCN is set to 0.005, while for the fully connected feedforward network is set to 0.001. The initial learning rate for the bi-LSTM is set to 0.001, and the training epochs are set to 60.

We employed four widely used evaluation metrics to gauge the effectiveness of our approach against other methods. Accuracy (Acc) measures the percentage of correctly identified samples. Precision (Pre) indicates the ratio of correctly identified vulnerable
samples to all detected vulnerable samples. Recall (Rec) represents the proportion of correctly identified vulnerable samples to all vulnerable samples. The F1-score (F1) serves as an overall effectiveness score and is calculated as the harmonic mean of precision and recall. The specific computation formulas for these evaluation metrics are as follows:

\[
\text{ACC} = \frac{TP + TN}{TP + FP + TN + FN} \tag{7}
\]

\[
R = \frac{TP}{TP + FN} \tag{8}
\]

\[
P = \frac{TP}{TP + FP} \tag{9}
\]

\[
F1 = 2 \times \frac{P \times R}{P + R} \tag{10}
\]

4.4. Comparative Experiments

In the first experiment, we investigated the detection performance of the proposed method for smart contract vulnerabilities.

Firstly, we compared our method with the single-modality approach (SC) competitors in terms of Acc, Rec, Pre, and F1. The performance results of different methods are listed in Table 4. From the table, we can observe the following:

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Method</th>
<th>Acc (%)</th>
<th>Rec (%)</th>
<th>Pre (%)</th>
<th>F1 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>SmartCheck</td>
<td>40.43</td>
<td>41.25</td>
<td>39.40</td>
<td>40.59</td>
</tr>
<tr>
<td></td>
<td>Slither</td>
<td>40.97</td>
<td>42.31</td>
<td>35.48</td>
<td>38.59</td>
</tr>
<tr>
<td></td>
<td>Mythril</td>
<td>51.88</td>
<td>49.69</td>
<td>52.30</td>
<td>50.98</td>
</tr>
<tr>
<td></td>
<td>CBGRU</td>
<td>85.43</td>
<td>84.25</td>
<td>85.41</td>
<td>84.83</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>85.73</td>
<td>86.54</td>
<td>84.85</td>
<td>85.69</td>
</tr>
<tr>
<td>Timestamp dependency</td>
<td>SmartCheck</td>
<td>58.94</td>
<td>77.45</td>
<td>37.45</td>
<td>50.44</td>
</tr>
<tr>
<td></td>
<td>Slither</td>
<td>76.27</td>
<td>89.47</td>
<td>77.27</td>
<td>82.96</td>
</tr>
<tr>
<td></td>
<td>Mythril</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>CBGRU</td>
<td>87.65</td>
<td>86.28</td>
<td>88.16</td>
<td>87.21</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>85.41</td>
<td>83.48</td>
<td>81.19</td>
<td>82.32</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>SmartCheck</td>
<td>41.12</td>
<td>69.34</td>
<td>43.69</td>
<td>53.60</td>
</tr>
<tr>
<td></td>
<td>Slither</td>
<td>60.96</td>
<td>82.09</td>
<td>62.97</td>
<td>71.27</td>
</tr>
<tr>
<td></td>
<td>Mythril</td>
<td>68.56</td>
<td>82.87</td>
<td>36.88</td>
<td>50.62</td>
</tr>
<tr>
<td></td>
<td>CBGRU</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>83.58</td>
<td>83.71</td>
<td>79.25</td>
<td>81.41</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>SmartCheck</td>
<td>58.70</td>
<td>30.30</td>
<td>33.33</td>
<td>31.74</td>
</tr>
<tr>
<td></td>
<td>Slither</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Mythril</td>
<td>75.34</td>
<td>77.94</td>
<td>74.64</td>
<td>75.82</td>
</tr>
<tr>
<td></td>
<td>CBGRU</td>
<td>86.54</td>
<td>87.23</td>
<td>85.66</td>
<td>86.43</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>90.96</td>
<td>91.62</td>
<td>89.16</td>
<td>90.50</td>
</tr>
</tbody>
</table>

From Table 4, it is evident that among these four tools, SmartCheck generally exhibits the lowest Acc and F1 scores. This is attributed to SmartCheck being an XML-based vulnerability detection tool. XML has limited expressive capabilities, failing to adequately reflect syntax and semantic information, thereby limiting the performance of vulnerability detection.

Slither performs better than SmartCheck, due to its use of a more detailed intermediate form called Static Single Assignment (SSA) to retain additional semantic information. Slither also leverages data flow and taint tracking to gain a deeper understanding of program behavior.
Mythril demonstrates higher detection performance due to its utilization of static symbolic execution technology, simulating the execution paths of smart contract code to identify potential paths leading to vulnerabilities. This approach aids in understanding contract execution logic and detecting potential vulnerability paths. However, compared to deep learning-based methods, Mythril still does not gain an advantage.

CBGRU is a vulnerability detection method based on source code and deep learning. Source code-based detection methods are often suitable for security vulnerabilities occurring within a single line of code. Experimental results demonstrate that CBGRU achieves the best detection accuracy for the timestamp dependency vulnerability. However, vulnerabilities such as integer overflow might directly reflect at the opcode layer, making them challenging to identify at the source code layer.

In most cases, our method outperforms the other methods. Specifically, our method improves the Acc for three out of four vulnerability types against the maximum value among the three competitors using traditional methods by 33.85%, 9.14%, 15.02%, and 15.62%, respectively. Additionally, for the F1 scores across the four vulnerability types, except for a slightly weaker performance compared to Slither in the case of timestamp dependency, our method significantly improves the F1 scores in the other scenarios. In the comparison with CBGRU, our method exhibits improvements in Acc and F1 scores for three out of four vulnerability cases, except for slightly weaker performance in detecting the timestamp dependency vulnerability. These improvements demonstrate the effectiveness of deep learning methods and the introduction of multimodal features.

Later, we compared our method with single-modality approach (EMVO) competitors in terms of Acc, Rec, Pre, and F1. The performance results of the different methods are listed in Table 5. From the table, we can observe the following:

Table 5. Experimental comparison with other methods.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Method</th>
<th>Acc (%)</th>
<th>Rec (%)</th>
<th>Pre (%)</th>
<th>F1 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>ContractFuzzer</td>
<td>38.90</td>
<td>68.01</td>
<td>32.54</td>
<td>44.22</td>
</tr>
<tr>
<td></td>
<td>Oyente</td>
<td>42.68</td>
<td>75.11</td>
<td>44.20</td>
<td>55.72</td>
</tr>
<tr>
<td></td>
<td>Securify</td>
<td>53.95</td>
<td>77.96</td>
<td>54.88</td>
<td>64.43</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>85.73</td>
<td>86.54</td>
<td>84.85</td>
<td>85.69</td>
</tr>
<tr>
<td>Timestamp dependency</td>
<td>ContractFuzzer</td>
<td>33.54</td>
<td>82.99</td>
<td>30.31</td>
<td>44.61</td>
</tr>
<tr>
<td></td>
<td>Oyente</td>
<td>45.23</td>
<td>77.82</td>
<td>39.28</td>
<td>52.21</td>
</tr>
<tr>
<td></td>
<td>Securify</td>
<td>52.06</td>
<td>84.32</td>
<td>51.21</td>
<td>63.71</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>85.41</td>
<td>83.48</td>
<td>81.19</td>
<td>82.32</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>ContractFuzzer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Oyente</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Securify</td>
<td>49.55</td>
<td>72.83</td>
<td>51.55</td>
<td>60.37</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>83.58</td>
<td>83.71</td>
<td>79.25</td>
<td>81.41</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>ContractFuzzer</td>
<td>33.56</td>
<td>58.44</td>
<td>30.36</td>
<td>40.01</td>
</tr>
<tr>
<td></td>
<td>Oyente</td>
<td>38.36</td>
<td>33.13</td>
<td>92.73</td>
<td>48.94</td>
</tr>
<tr>
<td></td>
<td>Securify</td>
<td>51.81</td>
<td>73.59</td>
<td>57.81</td>
<td>64.82</td>
</tr>
<tr>
<td></td>
<td>Our method</td>
<td>90.96</td>
<td>91.62</td>
<td>89.16</td>
<td>90.50</td>
</tr>
</tbody>
</table>

According to the data presented in Table 5, ContractFuzzer exhibits almost the worst detection performance across various scenarios. This is primarily due to the nature of fuzzing techniques, which require random inputs to trigger vulnerabilities. Consequently, its performance heavily relies on code coverage and test time.

Oyente, being a symbolic execution tool, outperforms ContractFuzzer. It can resolve code branches through constraint solving. However, it still has limitations when dealing with complex branches, such as the inability to resolve certain conditions’ exact values.

In comparison to the two aforementioned methods, Securify consistently demonstrates superior performance in various scenarios. This is credited to Securify’s adoption of a
formalized approach. However, creating a comprehensive formal specification covering the entire contract code is challenging.

Our method significantly outperformed the other three tools in terms of ACC and F1 scores across all scenarios. These improvements are attributed to the introduction of multimodal features and the powerful capabilities of the bi-LSTM model.

Given the ongoing evolution of vulnerabilities, relying solely on expert knowledge for traditional smart contract vulnerability detection might overlook crucial information necessary to identify these vulnerabilities. With the rapid expansion of decentralized applications based on blockchain technology, conventional methods might struggle to address the increasing number and diverse nature of unforeseeable smart contract vulnerabilities.

While deep learning-based methods have proven effective in handling a significant portion of current and potential smart contract vulnerabilities, these methods rely on a limited scope of contract feature information. Extracting more comprehensive information remains a crucial challenge.

Our approach aims to extract extensive contract feature information and seamlessly integrate these details into our model. Through this integrated information, we trained our model to accurately detect various types of smart contract vulnerabilities, thereby enhancing the security of blockchain applications. Consequently, our method outperformed existing methodologies in vulnerability detection.

4.5. Ablation Experiments

In this section, we validate the effectiveness of each component of the proposed method. First, we conducted experiments to compare the performance between single-modality features and the fused features. Next, we validated the effectiveness of multimodal, multi-type feature fusion. Finally, we verified the effectiveness of using the bi-LSTM model as the fusion model.

As shown in Table 6, it can be observed that using the new features from multiple modalities can effectively enhance the performance of the trained model compared to the original single-modality feature detection.

Table 6. Intermodal comparison.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Feature</th>
<th>Acc(%)</th>
<th>Rec(%)</th>
<th>Pre (%)</th>
<th>F1 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>SC</td>
<td>73.82</td>
<td>82.12</td>
<td>73.36</td>
<td>77.44</td>
</tr>
<tr>
<td></td>
<td>EVMO</td>
<td>73.55</td>
<td>65.48</td>
<td>76.39</td>
<td>70.51</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO</td>
<td>80.26</td>
<td>86.52</td>
<td>75.28</td>
<td>80.48</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO+DM</td>
<td>84.30</td>
<td>85.36</td>
<td>84.22</td>
<td>84.78</td>
</tr>
<tr>
<td>Timestamp</td>
<td>SC</td>
<td>81.78</td>
<td>78.81</td>
<td>76.32</td>
<td>77.54</td>
</tr>
<tr>
<td></td>
<td>EVMO</td>
<td>77.23</td>
<td>78.17</td>
<td>76.72</td>
<td>77.44</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO</td>
<td>78.54</td>
<td>79.46</td>
<td>77.90</td>
<td>78.67</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO+DM</td>
<td>84.72</td>
<td>82.60</td>
<td>80.74</td>
<td>81.66</td>
</tr>
<tr>
<td>Tx.origin</td>
<td>SC</td>
<td>81.18</td>
<td>78.38</td>
<td>81.56</td>
<td>79.94</td>
</tr>
<tr>
<td></td>
<td>EVMO</td>
<td>73.42</td>
<td>75.03</td>
<td>72.41</td>
<td>73.70</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO</td>
<td>81.16</td>
<td>76.19</td>
<td>85.49</td>
<td>80.58</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO+DM</td>
<td>82.96</td>
<td>80.41</td>
<td>82.64</td>
<td>81.51</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>SC</td>
<td>84.21</td>
<td>85.71</td>
<td>82.99</td>
<td>84.85</td>
</tr>
<tr>
<td></td>
<td>EVMO</td>
<td>87.33</td>
<td>87.50</td>
<td>90.92</td>
<td>89.20</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO</td>
<td>88.41</td>
<td>88.89</td>
<td>87.27</td>
<td>88.07</td>
</tr>
<tr>
<td></td>
<td>SC+EVMO+DM</td>
<td>89.31</td>
<td>90.05</td>
<td>88.26</td>
<td>88.69</td>
</tr>
</tbody>
</table>

As shown in Figure 10, to visually represent the effectiveness of the four feature extraction methods, this paper presents the model training results in graphical form:
The experimental results reveal notable differences within single modality setups. The SC modality, leveraging natural language processing techniques for feature extraction, demonstrates higher average accuracy and recall rates in detecting reentrancy, timestamp dependency, and tx.origin vulnerabilities compared to the EVMO modality. While the EVMO modality mainly focuses on extracting function call relations using graph-based feature extraction on opcodes, its information depth is relatively limited compared to the semantic richness provided by natural language processing, resulting in an overall inferior detection performance compared to the SC modality. Particularly, the EVMO modality excels in detecting the integer overflow vulnerability, primarily manifested at the opcode level, surpassing the efficiency of the SC modality in this aspect.

In the SC+EVMO dual-modality setup, we extract advanced semantic and structural information from smart contract source code alongside data dependency and control relation information. Consequently, the model’s detection performance in the SC+EVMO dual-modality setup surpasses that of the single-modality setup.

Furthermore, in the SC+EVMO+DM triple-modality setup, dynamic features, providing real-time runtime data and behavioral information, are integrated with static features that primarily focus on contract structure and attributes. This combined approach enables a more comprehensive and accurate contract analysis, identifying more potential vulnerabilities and reducing false-positive rates. The integration of pre-deployment static features with actual runtime data features covers different stages of the contract lifecycle, leading to superior model detection performance compared to both single and dual-modality setups. Hence, it can be concluded that multimodal feature fusion outperforms single-modality setups, and models incorporating dynamic features perform better than those using only static features.

In addition, during the fusion model training phase, we replaced the bi-LSTM model with RNN, GRU, and LSTM for performance comparison in modal fusion. The use of the bi-LSTM model significantly improves the performance of smart contract vulnerability detection. Taking the integer overflow vulnerability as an example and comparing the effectiveness of vulnerability detection under different models, the experimental results are shown in Table 7.

Figure 10. Performance evaluation of different types of vulnerabilities under different modal settings. In the figure, the x-axis represents different evaluation metrics; the y-axis represents the score of a specific evaluation metric, ranging from 0.00% to 100.00%.
Table 7. Comparison between different training models.

<table>
<thead>
<tr>
<th>Method</th>
<th>Acc (%)</th>
<th>Rec (%)</th>
<th>Pre (%)</th>
<th>F1 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td>83.76</td>
<td>86.21</td>
<td>83.39</td>
<td>84.78</td>
</tr>
<tr>
<td>GRU</td>
<td>81.60</td>
<td>80.71</td>
<td>83.47</td>
<td>82.07</td>
</tr>
<tr>
<td>LSTM</td>
<td>88.18</td>
<td>90.34</td>
<td>86.96</td>
<td>88.62</td>
</tr>
<tr>
<td>bi-LSTM</td>
<td>90.96</td>
<td>91.62</td>
<td>89.16</td>
<td>90.50</td>
</tr>
</tbody>
</table>

5. Results

The experimental outcomes indicate the superiority of the proposed intelligent contract vulnerability detection method over existing approaches, showcasing high accuracy and recall rates in detecting vulnerabilities. Ablation experiments confirm the enhanced performance of the vulnerability detection model upon the inclusion of new modal features. Additionally, the amalgamation of dynamic and static features during model training proves beneficial in augmenting detection accuracy for intelligent contract vulnerability detection tasks. In these detection tasks, prioritizing high recall is more crucial than high precision due to the higher cost associated with false negatives than false positives. Overall, the experimental results underscore the robust performance of the proposed multi-modal deep learning approach in detecting smart contract vulnerabilities, significantly surpassing traditional methods and single-modal feature extraction approaches.

6. Conclusions

In summary, this study proposes a smart contract vulnerability detection method based on multimodal feature fusion. Taking a multimodal perspective, the method extracts three modal features from the entire lifecycle of smart contracts, effectively combining static and dynamic features. Through the utilization of deep learning models like GCN and bi-LSTM, efficient detection of smart contract vulnerabilities is achieved. Experimental results demonstrate outstanding performance of our method in vulnerability detection, exhibiting significant advantages over existing approaches. The ablation experiments confirmed the effectiveness of the new modal features and demonstrated that integrating dynamic and static features enhances the accuracy and comprehensiveness of smart contract vulnerability detection. This fusion reduces false positives and better addresses complex security threats, playing a crucial role in enhancing the security and reliability of contracts.

This research provides an effective solution for smart contract vulnerability detection with broad potential applications. When compared to existing tools for smart contract vulnerability detection, this method covers a wider range of vulnerability types and achieves higher detection accuracy. The proposed approach can be employed for the initial screening of smart contract vulnerabilities, enhancing the efficiency of smart contract developers and auditors in identifying vulnerabilities.

However, our method has several limitations. This study solely validated the effectiveness of the prototype system using four specific vulnerability types, whereas smart contract vulnerabilities are notably more diverse and can originate from various layers within the blockchain system—ranging from the blockchain system itself to the EVM execution layer and high-level programming language code. Moreover, our model’s scope is restricted to a single platform and language (Ethereum and Solidity), lacking broader applicability to other smart contract development languages such as Vyper, Serpent, or Bamboo. Additionally, while our model requires Solidity source code as an input, many smart contracts are publicly available only in bytecode format, rendering the extraction of their source code-level features unfeasible. Furthermore, the dynamic matching patterns developed in this study were tailored to known vulnerabilities. The identification of new dynamic matching patterns for additional and future types of vulnerabilities requires further research and design efforts. Lastly, the absence of standardized and unified smart contract vulnerability datasets remains a pressing issue in experiments. The dataset utilized was generated based on markings from static analysis tools, yet the presence of false positives in these tools undermines the authenticity of smart contract labels.
Future research directions may delve deeper into optimizing multimodal feature fusion methods by embracing a broader spectrum of feature types to enhance the accuracy and resilience of vulnerability detection. Potential avenues of exploration could involve the introduction of more intricate deep neural network architectures or leveraging attention mechanisms to integrate information more effectively. Additionally, expanding this methodology to encompass various other vulnerability types, devising novel matching patterns, gathering diverse sets of vulnerable contract data, and training models proficient in detecting multiple vulnerability types could be avenues for investigation. Moreover, collaborating with real-world smart contract platforms and developers to implement vulnerability detection methods in practical settings could validate the method’s efficacy in real-world scenarios. Such collaboration can provide valuable insights into the method’s practical feasibility and its performance in real applications. Considering the evolution of the blockchain ecosystem, where smart contracts transcend single platforms, future research could pivot towards cross-platform and cross-chain smart contract vulnerability detection to extend the method’s applicability. Lastly, constructing standardized datasets through manual labeling techniques using static analysis tools can ensure the authenticity of contract labels. This process contributes significantly to maintaining the validity and reliability of datasets utilized for smart contract analysis.

**Author Contributions:** Conceptualization, J.L.; methodology, J.L.; software, J.L.; validation, J.L.; formal analysis, F.G.; investigation, Y.G.; data curation, J.L.; writing—review and editing, G.L.; visualization, Y.G.; project administration, G.L.; funding acquisition, F.G. All authors have read and agreed to the published version of the manuscript.

**Funding:** This paper was financially supported by the project of Research and Application Demonstration of Key Technologies of Yunnan Autonomous Controllable Blockchain Basic Service Platform (grant no. 202102AD080006).

**Data Availability Statement:** No new data were created or analyzed in this study. Data sharing is not applicable to this article.

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

**References**

5. Zhang, Y.; Liu, D. Toward vulnerability detection for ethereum smart contracts using graph-matching network. *Future Internet* 2022, 14, 326. [CrossRef]


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.