electronics

Article

A Smart Contract Vulnerability Detection Method Based on Heterogeneous Contract Semantic Graphs and Pre-Training Techniques

Jie Zhang, Gehao Lu * and Jia Yu

School of Information Science and Engineering, Yunnan University, Kunming 650500, China; zjlyy@mail.ynu.edu.cn (J.Z.); yujia@itc.ynu.edu.cn (J.Y.)
* Correspondence: glu@ynu.edu.cn

Abstract: The use of smart contracts in areas such as finance, supply chain management, and the Internet of Things has significantly advanced blockchain technology. However, once deployed on the blockchain, smart contracts cannot be modified or revoked. Any vulnerabilities can lead to severe economic losses and data breaches, making pre-deployment vulnerability detection critically important. Traditional smart contract vulnerability detection methods suffer from low accuracy and limited reusability across different scenarios. To enhance detection capabilities, this paper proposes a smart contract vulnerability detection method based on heterogeneous contract semantic graphs and pre-training techniques. Compared to the conventional graph structures used in existing methods, heterogeneous contract semantic graphs contain richer contract information. By integrating these with pre-trained models, our method exhibits stronger vulnerability capture and generalization capabilities. Experimental results show that this method has improved the accuracy, recall, precision, and F1 value in the detection of four widely existing and harmful smart contract vulnerabilities compared with existing methods, which greatly improves the detection ability of smart contract vulnerabilities.

Keywords: smartcontracts; vulnerability detection; heterogeneous contract semantic graphs; pre-training techniques

check for updates

Citation: Zhang, J.; Lu, G.; Yu, J. A Smart Contract Vulnerability Detection Method Based on Heterogeneous Contract Semantic Graphs and Pre-Training Techniques. *Electronics* **2024**, *13*, 3786. https://doi.org/10.3390/electronics13183786

Academic Editor: Fabio Grandi

Received: 31 July 2024
Revised: 17 September 2024
Accepted: 20 September 2024
Published: 23 September 2024

1. Introduction

Blockchain technology, since it was proposed by Satoshi Nakamoto in the Bitcoin white paper in 2008 [1], has rapidly emerged as a disruptive technology with potential applications in various fields, including finance, supply chain management, and the Internet of Things (IoT). Ethereum, as the second-generation blockchain platform, introduced smart contracts as a core feature, enabling decentralized computation on top of the blockchain [2], which significantly expanded the scope of blockchain applications.

The automation and immutability of smart contracts have increased trust in their use across various sectors; however, these same characteristics present significant security challenges. Once vulnerabilities are embedded in smart contract code, they can lead to major losses for associated accounts. According to the SlowMist Technology report [3], there were 464 security incidents in 2023, resulting in losses totaling 2.486 billion USD. Notable cases include the non-custodial lending platform BonqDAO and the crypto infrastructure platform AllianceBlock, which lost approximately 120 million USD due to vulnerabilities in BonqDAO's smart contracts. Therefore, comprehensive security testing of smart contracts before deployment is crucial.

Currently, researchers have developed many effective tools for detecting vulnerabilities in smart contracts, which can be divided into two main categories: traditional methods, such as Manticore [4], SmarTest [5], ConFuzzius [6], Slither [7], and deep learning-based methods, such as VanillaRNN [8], LSTM [9], TMP [10]. However, these methods have certain limitations:

- traditional methods rely on complex predefined patterns or meticulously designed test cases, requiring experts to deeply analyze vulnerabilities and continually update specific rules. These methods struggle to scale with the rapid growth of smart contract deployments, which restricts their applicability and leads to lower accuracy.
- Deep learning-based methods often inadequately consider the structural and semantic information of code, leading to insufficient generalization when facing diverse smart contracts.

In order to address the shortcomings of existing methods, we select four smart contract vulnerabilities that cause the most serious and widespread losses: Reentrancy [11], Transaction State Dependency [12], Block Info Dependency [13], and Nested Call [12] as research objects. These vulnerabilities were chosen based on the research findings of Monika [12] and Chen [14], who categorized and graded the impact of smart contract vulnerabilities. Additionally, their importance has been validated through numerous studies (e.g., [15,16]) and widely recognized classifications, such as the SWC Registry [13]. See Appendix A for corresponding code examples and descriptions. This paper proposes a smart contract vulnerability detection method based on heterogeneous contract semantic graphs and pre-training techniques. Compared with the traditional data flow graph used in existing methods, the heterogeneous contract semantic graph contains richer contract structure information, which enables the model to focus on the key features of the vulnerability more comprehensively and has stronger vulnerability capture and generalization capabilities. The heterogeneous contract semantic graph is embedded in the graph neural network so that it can be used as part of the pre-training model input to detect smart contract vulnerabilities.

The main contributions of this paper are as follows:

- We designed a heterogeneous smart contract semantic graph generation method based on abstract syntax trees (AST), using variables and function calls as nodes to comprehensively represent the semantic and structural information of contracts at the statement level.
- By combining heterogeneous contract semantic graphs with pre-training techniques, our Approach achieved superior performance in detecting various types of smart contract vulnerabilities. Experimental results show that our method achieves detection accuracies of 90.96%, 89.57%, 88.46%, and 87.34% for the four types of vulnerabilities, representing significant improvements over 12 mainstream methods.

Next, Section 2 will introduce the existing work on smart contract vulnerability detection and the relevant background knowledge of this article. Section 3 will introduce the smart contract vulnerability detection method based on heterogeneous contract semantic graphs and pre-training technology. Section 4 will show the experimental results, and Section 5 will summarize.

2. Related Work

2.1. Existing Smart Contract Vulnerability Detection Tools

With the rapid development of blockchain platforms such as Ethereum, numerous smart contract vulnerability detection tools have emerged. In the following, we will briefly introduce mainstream methods for detecting smart contract vulnerabilities. We will also analyze their strengths and limitations. Representative traditional methods include those based on symbolic execution (e.g., Oyente [17], Manticore [4], SmarTest [5], Mythril [11]), fuzz testing (e.g., ContractFuzzer [18], ConFuzzius [6]), static analysis (e.g., Slither [7], SmartCheck [19]), and formal verification (e.g., KEVM [20], ZEUS [21]).

Symbolic execution-based methods, such as Oyente [17], use symbolic inputs instead of concrete ones. They simulate the execution of the analyzed program and transform program operations into symbolic expressions. This allows for the analysis of path reachability, test data generation, and detection of specific vulnerabilities. However, as the size of smart contract code increases, these methods may face challenges like path explosion and slow constraint solving. This can reduce the efficiency of vulnerability detection. Fuzz testing-based methods, such as ContractFuzzer [18], generate many random test cases by obtaining function parameters through the Abstract Binary Interface (ABI). They combine these cases with vulnerability patterns to detect issues like reentrancy vulnerabilities. However, these test cases often lack specificity and may not cover all possible vulnerability scenarios. For complex smart contracts, fuzz testing may fail to trigger deeply hidden vulnerabilities, especially those that only appear under specific input conditions. Slither [7], as a static analysis tool, directly analyzes the source code of smart contracts. It can quickly detect potential vulnerabilities without executing the contract. This makes it more efficient in speed, allowing it to handle large-scale smart contract codebases. However, static analysis tools typically rely on predefined rules or patterns. As a result, Slither may fail to detect certain atypical vulnerabilities, especially those involving complex state changes or dynamic behaviors. KEVM [20] uses formal modeling of smart contract behavior to verify whether a contract adheres to specific security properties. It provides a high level of security assurance. However, formal verification involves exhaustive checking of all possible execution paths. As a result, KEVM's verification process often requires significant time. This can be problematic when handling complex contracts, leading to excessively long verification times.

In recent years, deep learning has been widely applied in various fields. Leveraging big data and deep learning techniques to automatically learn the characteristics of smart contract vulnerabilities has become a new research direction. Wang et al. [15] developed ContractWard, which extracts features from bytecode using n-grams. The model uses five machine learning algorithms, including Random Forest and Support Vector Machine, to detect six types of vulnerabilities, such as reentrancy and integer overflow. However, this method does not consider the structural or semantic information of the code during learning and computation. Liu et al. [22] proposed the vulnerability detection network CGE (Combining Graph features and Expert patterns). It defines expert pattern features, such as timestamp declarations, and combines them with contract graph features obtained through graph neural networks. This improves the accuracy of vulnerability detection. However, their model cannot handle heterogeneous graphs with multiple types of nodes. They addressed this by removing secondary nodes and aggregating their features into primary nodes, which led to the loss of some information. This reduced the method's generalizability when dealing with diverse smart contracts.

2.2. Graph Pre-Trained Models

Pre-trained models begin by undergoing preliminary training on large-scale, unsupervised datasets to learn general features and representations. They are then fine-tuned on smaller, labeled datasets to adapt to specific tasks. The primary advantage of this approach is the ability to leverage the extensive information from large datasets to enhance performance on specific tasks. Models like BERT [23] and GPT [24] have achieved notable success in natural language processing tasks and are widely used. Inspired by this, researchers in the field of software engineering have proposed a series of models specifically for programming languages, such as CodeBERT [25] and CodeT5 [26], which are designed to understand and generate source code. Similar to text-based pre-training models, these models learn the structure, syntax, and common programming patterns by pre-training on large amounts of open-source code, significantly improving the automation and efficiency of programming-related tasks.

However, some existing works [25,26] treat program source code as collections of tokens, overlooking that code is not merely a sequence of words but includes various graph-like data structures such as loops, jumps, controls, and dependencies [27]. GraphCodeBERT [28] makes up for the shortcomings of previous work by incorporating semantic-level information about the code and introducing the data flow graph (DFG) of the code for pre-training. The data flow graph is a structure frequently used in program analysis [29]. In GraphCodeBERT, code is first parsed into an Abstract Syntax Tree (AST) using standard compilation tools. From the AST, a labeled sequence of source code variables is derived, and a data flow graph is constructed. This graph represents program variables as nodes and the dependencies among them as edges. The code and its corresponding data flow graph serve as inputs to the model, achieving good results in a variety of code-related downstream tasks.

2.3. Graph Structure Representation of Smart Contracts

Current research [29] suggests that programs can be transformed into graphical representations, thereby preserving the relationships between program elements. Zhuang et al. [10] have utilized this approach by converting smart contracts into graphs and employing Graph Neural Networks (GNN) for detecting vulnerabilities in smart contracts. However, their model struggles with handling heterogeneous graphs with multiple types of nodes, leading them to remove secondary nodes and aggregate their features into primary nodes, resulting in a partial loss of structural information. Wu et al. [30] introduced a method for smart contract vulnerability detection named Peculiar, which uses the tree-sitter-solidity tool https://github.com/JoranHonig/tree-sitter-solidity (accessed on 31 July 2024) to generate an AST. From the AST, they extract a Data Flow Graph (DFG) to represent the contract's graphical structure and have designed a method to filter key variables, simplifying the DFG into a Critical Data Flow Graph (CFDG) for model input. Although the CDFG simplifies smart contract information to some extent and focuses more on variables that could lead to vulnerabilities, it only includes the data flow relationships between variables and misses a lot of structural information related to vulnerabilities, such as special function calls and execution sequences. In order to solve the problem that the existing smart contract graph structure cannot adequately represent smart contract information, this paper proposes a new smart contract graph structure representation method, which uses variables and function calls as nodes to construct a heterogeneous contract graph. Based on this heterogeneous graph, we further consider how to effectively capture the complex semantic relationships within the graph. There are several embedding methods for heterogeneous graphs, such as metapath2vec [31], HAN [32], and HGT [33]. Metapath2vec defines meta-paths to guide random walks, capturing semantic relationships in heterogeneous graphs, and then uses the Skip-Gram model to learn node embeddings. HAN utilizes meta-paths to guide attention mechanisms, focusing on interactions between nodes with significant semantic importance. HGT, based on the Transformer architecture, introduces type-specific parameterization to handle the heterogeneity of nodes and edges, and incorporates relative temporal encoding to account for the dynamic nature of the graph. This is particularly crucial for smart contract vulnerability detection, as vulnerabilities are often closely related to data flow and control flow timing between nodes. Therefore, we select HGT as the graph embedding method to fully represent the semantic information and structural relationships of smart contracts.

3. Research Method

In the methodology of this paper, we do not undertake the pre-training task ourselves; instead, the pre-training is handled by GraphCodeBERT [28]. This is appropriate because the Solidity language, specifically designed for Ethereum smart contract development, exhibits features like static typing, inheritance, libraries, and complex user-defined types, drawing influences from JavaScript, Python, and C++. GraphCodeBERT, during its pre-training phase, utilized 2.3 million functions from six programming languages, encompassing three pre-training tasks: modeling programming languages, predicting edges in the program data graph (data flow graph), and aligning variables across program source code and its graph structure. Our method requires loading the pre-trained parameters from GraphCodeBERT, followed by fine-tuning them on a smart contract dataset. The workflow of this method is shown in Figure 1 and is divided into two stages : (a) Graph Generation Stage, during which source code is converted into an AST, parsing the AST to retain different nodes and edges and generating a heterogeneous contract semantic graph. (b) Vulnerability Detection Stage, where smart contract vulnerabilities are detected based on the pre-trained model. We will now elaborate on these two stages.

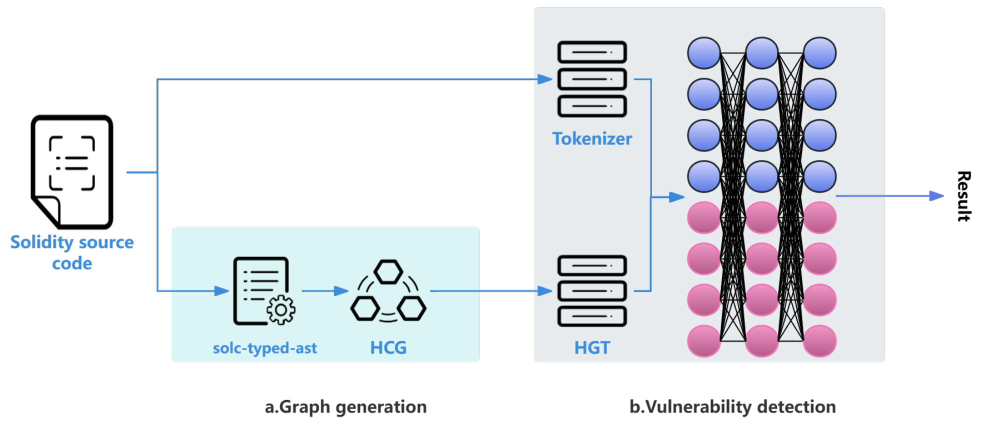
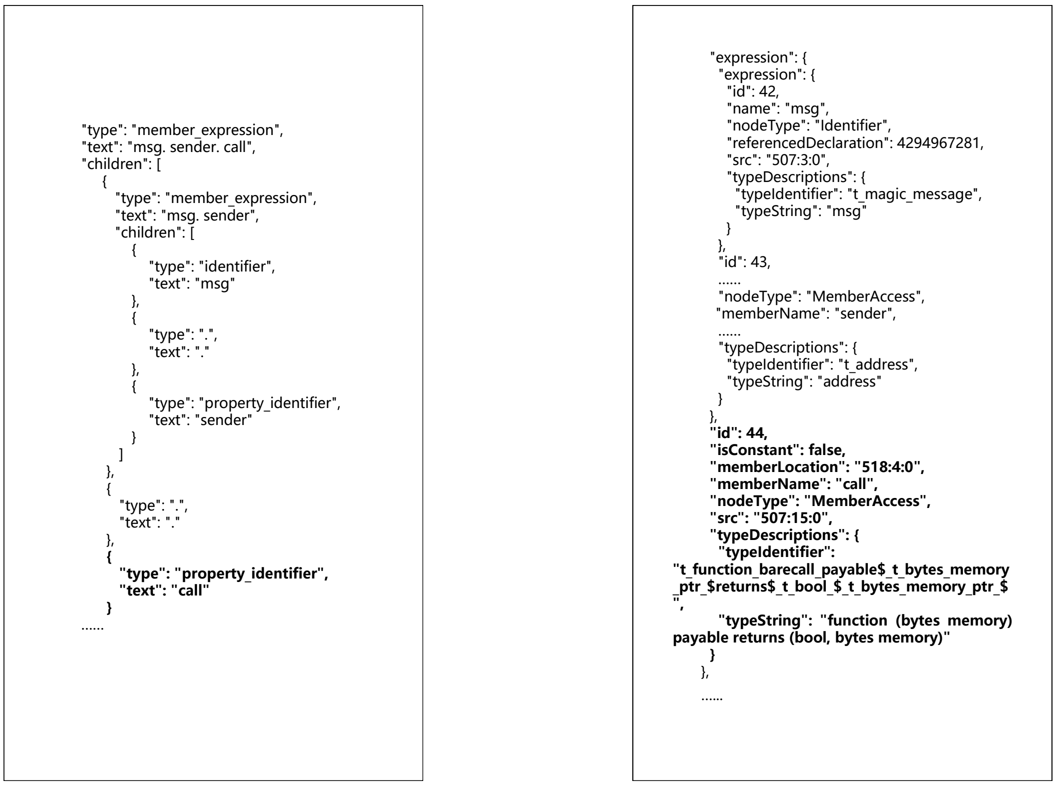


Figure 1. Method workflow. (**a**) Graph generation phase: The smart contract source code is first generated into a standardized syntax tree by the solc-typed-ast tool. Based on the information in the standardized syntax tree, the variables and function call behaviors in the source code are used as nodes to construct a heterogeneous contract semantic graph. (**b**) Vulnerability detection phase: HGT is used to embed the semantic graph of heterogeneous smart contracts, and the corresponding source code is used as the input of GraphCodeBERT to detect vulnerabilities in smart contracts.

3.1. Graph Generation Phase

In the process of generating the AST for Solidity code, most existing work, such as [10,30], predominantly employs the tool tree-sitter-solidity, which is known for its efficiency and incremental parsing capabilities. However, for Solidity code, the AST generated by this tool does not contain sufficiently detailed node information. For example, as shown in Listing A1, concerning the content at line 11 of the reentrancy vulnerability example code *msg.sender.call*, Figure 2a represents a portion of the AST nodes obtained from tree-sitter-solidity, whereas Figure 2b shows a portion of the AST nodes parsed by the official Solidity compiler, solc. It is evident that the AST nodes parsed by tree-sitter-solidity only include type and text information within their description, lacking detailed metadata. In contrast, the AST generated by solc not only includes structural information within the AST hierarchy but also provides rich descriptive information for nodes such as id, src, nodeType, and typeDescriptions, particularly offering standardized descriptions in typeIdentifier and typeString for critical nodes associated with vulnerabilities like the low-level function call *call()*. Given that solc has undergone multiple iterations since 2015, the AST structure and node information generated by different versions of solc can vary. The tool solc-typed-ast enables the creation of normalized type Solidity ASTs based on solc, mitigating the differences brought by various versions of the compiler. Therefore, we opt to use solc-typed-ast to generate the code AST.

The AST generated by the solc-typed-ast tool includes 56 types of nodes, which we categorize into three classes based on their functionalities. They are structure operation class nodes, variable and call class nodes, and other class nodes. The detailed classification is shown in Table A1 in Appendix A. The structural and operational class nodes encompass the structure and control information of an entire '.sol' file's AST, from top to bottom, including files, contracts, functions, statements, and operations. We utilize this information to construct the edges in the heterogeneous contract semantic graph, including both data flows and control flows. The variable and call class nodes, as components of statements, are constructed as nodes within the heterogeneous contract semantic graph. Other class nodes, which include import statements, structured documentation, and other nodes that do not contribute to vulnerability generation, or basic class nodes like names, contain information that can be encapsulated by higher-level nodes and may be omitted.



```
"type": "member_expression",
"text": "msg. sender. call",
"children": [
    {
      "type": "member_expression",
      "text": "msg. sender",
      "children": [
          {
              "type": "identifier",
              "text": "msg"
          },
          {
              "type": ".",
              "text": "."
          },
          {
              "type": "property_identifier",
              "text": "sender"
          }
      ]
    },
    {
      "type": ".",
      "text": "."
    },
    {
      "type": "property_identifier",
      "text": "call"
    }
......
```

```
"expression": {
  "expression": {
    "id": 42,
    "name": "msg",
    "nodeType": "Identifier",
    "referencedDeclaration": 4294967281,
    "src": "507:3:0",
    "typeDescriptions": {
      "typeIdentifier": "t_magic_message",
      "typeString": "msg"
    }
  },
  "id": 43,
  ......
  "nodeType": "MemberAccess",
 "memberName": "sender",
  ......
  "typeDescriptions": {
    "typeIdentifier": "t_address",
    "typeString": "address"
  }
},
"id": 44,
"isConstant": false,
"memberLocation": "518:4:0",
"memberName": "call",
"nodeType": "MemberAccess",
"src": "507:15:0",
"typeDescriptions": {
  "typeIdentifier":
"t_function_barecall_payable$_t_bytes_memory
_ptr_$returns$_t_bool_$_t_bytes_memory_ptr_$
",
  "typeString": "function (bytes memory)
payable returns (bool, bytes memory)"
 }
},
......
```

(**a**) tree-sitter-solidity Parsing result (**b**) solc Parsing results

Figure 2. The information contained in the AST obtained by the two tools for parsing the same statement is very different. For a more intuitive comparison, we bold the parsing results of the same element "call".

Specifically, in high-level programming languages, the basic units of execution are statements, including both simple and compound statements. Simple statements consist of a single logical line, such as expression statements (ExpressionStatement) and assignment statements (Assignment), whereas compound statements encompass other statements (groups of statements) that influence or control the execution of the included statements in some manner, such as if statements (IfStatement) and for statements (ForStatement). The logic within simple statements facilitates operations on variables and function calls. In our proposed method for constructing heterogeneous contract semantic graphs, we analyze each simple statement in the contract according to the typed-AST structure, extracting variables and function call nodes to serve as nodes within the heterogeneous contract semantic graph. It is important to note that Solidity includes certain special variables that are always present in the global namespace. These built-in variables are essentially of a basic type and are used to describe attributes of blocks and transactions, such as *block.prevrandao* and *msg.sender* [34]. Although these variables are not explicitly declared, they are still integrated as nodes in our graph. Solidity allows for the manipulation of mappings, arrays, and structures through dereferencing, but we do not treat the results of such dereferences (e.g., *balances[msg.sender]*) as individual nodes. Instead, we regard these structures as a whole, which facilitates their inclusion as nodes when constructing the contract semantic graph.

The relationships among simple statements—such as sequence, loops, and conditionals—act as control flows, while the operations within simple statements (such as BinaryOperation, IndexAccess, etc.) and the inputs and outputs involved in function calls are constructed as data flows. Take the code shown in Listing 1 as an example:

Listing 1. Sample Code.

```
contract ReentrantBank {
mapping(address => uint256) public balances;
function deposit() public payable {
require(msg.value > 0);
balances[msg.sender] += msg.value;
}
function withdraw(uint256 _amount) public {
require(balances[msg.sender] >= _amount);
(bool sent, ) = msg.sender.call{value: _amount}("");
require(sent, "Failed to send Ether");
balances[msg.sender] -= _amount;
}
```

The visualization of the typed-AST generated by the solc-typed-ast tool is shown in Figure 3.

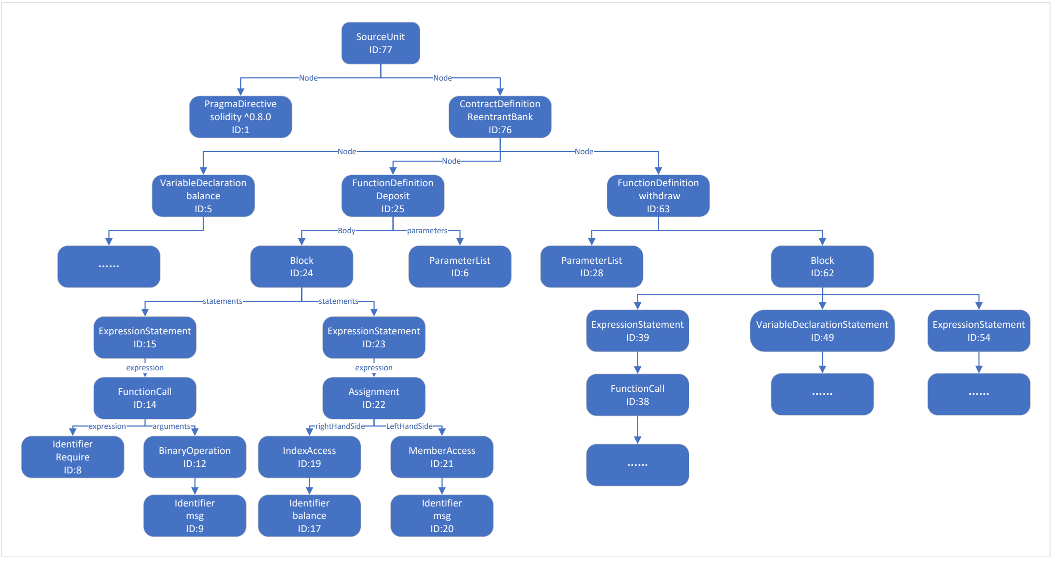
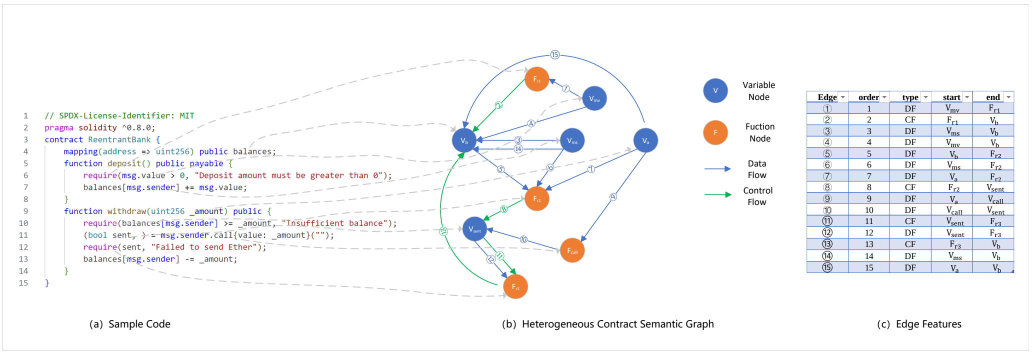


Figure 3. Typed–AST visualization.

Based on the structural information in the typed-AST, the example code is transformed into a heterogeneous contract semantic graph as shown in Figure 4 .

The variable node features in the contract graph are represented by a four-tuple, F = (id, name, nodeType, typeDescriptions). The function call node features in the contract graph are represented by a five-tuple, F = (id, name, nodeType, typeDescriptions, parameter), where id is the node identifier, name is the variable name (function name), nodetype is the node type, typeDescriptions is the specific description of the node type, and parameter is the function parameter. The edge features in the contract graph are represented by a four-tuple, F = (order, type, V_{start}, V_{end}), where order is the time order, type is the edge type (control flow or data flow), and V_{start} and V_{end} are the start and end nodes. The resulting contract graph is denoted as $HCG(SC) = (V, E)$, where V is the node set and E is the edge set. Figure 4b shows the heterogeneous contract semantic graph built based on the contract with the vulnerability, and Figure 4c shows its edge features.


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract ReentrantBank {
    mapping(address => uint256) public balances;
    function deposit() public payable {
        require(msg.value > 0, "Deposit amount must be greater than 0");
        balances[msg.sender] += msg.value;
    }
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");
        balances[msg.sender] -= _amount;
    }
}
```

Edge	order	type	start	end
①	1	DF	V_{mv}	F_{r1}
②	2	CF	F_{r1}	V_b
③	3	DF	V_{ms}	V_b
④	4	DF	V_{mv}	V_b
⑤	5	DF	V_b	F_{r2}
⑥	6	DF	V_{ms}	F_{r2}
⑦	7	DF	V_a	F_{r2}
⑧	8	CF	F_{r2}	V_{sent}
⑨	9	DF	V_a	V_{call}
⑩	10	DF	V_{call}	V_{sent}
⑪	11	CF	V_{sent}	F_{r3}
⑫	12	DF	V_{sent}	F_{r3}
⑬	13	CF	F_{r3}	V_b
⑭	14	DF	V_{ms}	V_b
⑮	15	DF	V_a	V_b

Figure 4. Heterogeneous Contract Semantic Graph Generation Process, Among them, (**a**) is the sample code in Listing 1, (**b**) is the heterogeneous contract semantic graph generated based on sample code, and (**c**) is the edge information in the heterogeneous contract semantic graph.

3.2. Vulnerability Detection Phase

This section will offer an in-depth description of how the proposed approach utilizes a graph-based pre-trained model to identify vulnerabilities in smart contracts. It will encompass details on data preparation, the model's architecture, and the training procedure.

First is the data preparation process. This method is based on GraphCodeBERT [28], but it differs in how it handles graph-structured data. In the original GraphCodeBERT architecture, given a source code $SC = \{sc_1, sc_2, \ldots, sc_n\}$, a corresponding data flow graph $G(SC) = (V, E)$ can be obtained, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of variables (also the nodes in the data flow graph), and $E = \{e_1, e_2, \ldots, e_n\}$ is a set of directed edges indicating the dependencies among variables. The source code and the set of variables are merged into a sequence $I = \{[CLS], SC, [SEP], V\}$, with $[CLS]$ as a special token preceding the sets, and $[SEP]$ as a separator between the source code SC and the variable set V. For each token in the sequence I, we generate the corresponding position embedding and add the token and the corresponding position embedding to represent the token. Special position embeddings are assigned to all variables to signify their roles as nodes within the data flow. The final input representation, denoted as X_0, is derived in this manner.

In our method, the embedding for the source code text follows the approach used in GraphCodeBERT. However, the embedding for the graph structure differs because the GraphCodeBERT method, which is suited for handling simple homogeneous graphs like data flow graphs, does not adequately capture the information in the heterogeneous contract semantic graph $HCG(SC) = (V, E)$ generated earlier. In our graph embedding process, we choose HGT [33] as the graph structure embedding method to extract the structural information of the heterogeneous contract semantic graph. The resulting node feature sequence replaces the variables sequence in the GraphCodeBERT method.

The purpose of HGT is to consolidate information from source nodes to obtain the contextual representation of target nodes. This procedure can be segmented into three distinct phases:

- The first part is the calculation of heterogeneous mutual attention. The calculation of heterogeneous mutual attention begins by examining the meta-relationships between a target node t and each of its source nodes $s \in N(T)$. These relationships are defined by the tuple $\langle T(s), \phi(e), T(t) \rangle$, representing the source node type, the edge type, and the target node type, respectively. To accommodate the diverse and complex nature of these relationships in a heterogeneous graph, the model converts target node t into a query vector and each source nodes s into a key vector. Unlike standard Transformers that use a direct inner product for such calculations, HGT utilizes distinct attention matrices $W^{ATT}_{\phi(e)}$ tailored to each edge type $\phi(e)$, ensuring that the nuances of different semantic associations are captured effectively.
- The second part is the heterogeneous message-passing process. The message-passing process from the source node to the target node and the calculation of mutual attention are parallel. The goal is to merge the meta-relationships of various edges into the

message-passing process. By doing so, it helps to balance the distribution disparities among different types of nodes and edges.

- The third part is the aggregation for a specific task. It uses the attention vector as the weight to calculate the corresponding information from the source node and obtain the updated vector. The updated vector is linearly mapped and connected with the original vector of t in the previous layer as a residual. In this way, the output $H^{(L)}[t]$ of the target node t in the L_{th} layer of HGT is obtained. Stacking L layers can obtain a rich context representation $H^{(L)}$ for each node as the input of the downstream task.

In order to be able to handle the dynamic nature of the graph, relative time coding is introduced. Traditionally, time information has been integrated by constructing a separate graph for each time slot, a method that can lead to the loss of structural dependencies across different time slots. Moreover, the representation of a node at time t might rely on edges from various other time slots. Thus, the appropriate method to model a dynamic graph is to preserve all edges occurring at various times and permit interactions between nodes and edges that possess different timings [32]. Specifically, given a source node s and a target node t with the edge $\phi(e)$ between them , the edge information $\phi(e)$ includes temporal information order, which is used as an index for the relative temporal encoding. This encoding is applied through sine and cosine functions to capture the relative temporal dependencies.

$$\textit{Base(order, 2i)} = \sin\left(\frac{\textit{order}}{10000^{\frac{2i}{d}}}\right) \tag{1}$$

$$\textit{Base(order, 2i+1)} = \cos\left(\frac{\textit{order}}{10000^{\frac{2i+1}{d}}}\right) \tag{2}$$

$$\textit{RTE(order)} = \textit{Linear(Base(order))} \tag{3}$$

In Equations (1) and (2), $order$ is the timing information carried by the edge in the heterogeneous graph, i is the position index, and d is the dimension of the feature vector in HGT. $Base$ represents the basic relative temporal encoding calculated by sine and cosine functions. In Equation (3), $Linear$ represents the projection of the basic relative time code to obtain the final relative time code RTE(order). RTE(order) is added to the node representations $H^{(L)}$ to obtain $H^{(L)'}$. This allows the resulting node representations to capture the relative temporal information between the source node s and the target node t.

Secondly, the model architecture and training process are as follows. As shown in Figure 5, the sequence $I = \{[CLS], SC, [SEP], H^{(L)'}\}$ obtained in the data preparation stage is fed to the Join layer. In the Join layer, the sequence I from the data preparation stage is converted into an input vector X_0. Subsequently, the input vector X_0 proceeds through the multi-head attention layer, undergoes layer normalization, and passes through multiple Transformer layers ($n = 12$) to produce distinct contextual representations, $X^n = \text{Transformer}_n(X^{n-1}), n \in (1, 12)$. Equations (4) and (5) represent the training process of the model. In Equation (4), H and X are vectors, $MultiAttn$ denotes a multi-head self-attention operation, and LN denotes a layer normalization operation. In Equation (5), FNN represents a two-layer feed-forward network, where each Transformer layer comprises a structurally identical transformer. As shown in Equations (4) and (5), the output X^{n-1} of the previous layer first undergoes a multi-head self-attention operation. The output of the self-attention operation is not directly passed to the next stage, but is first added to X^{n-1} to form a residual connection and normalized to obtain a vector H^n. After the vector H^n passes through the feedforward layer, which includes two linear transformation layers with an activation function in between, it also undergoes a residual connection and another layer of normalization to produce the output X^n. This process helps the model avoid potential gradient vanishing problems in deep networks while maintaining information from each layer's input.

$$H^n = LN(MultiAttn(X^{n-1}) + X^{n-1}) \tag{4}$$

$$X^n = LN(FFN(H^n) + H^n) \tag{5}$$

For the output $\hat{X}$ of the multi-head self-attention in the n_{th} transformer layer, the calculation process is shown in Equation (6) to (9):

$$Q_i = X^{(n-1)}W_i^Q, \quad K_i = X^{(n-1)}W_i^K, \quad V_i = X^{(n-1)}W_i^V \tag{6}$$

$$head_i = Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M\right)V_i \tag{7}$$

$$M_{ij} = \begin{cases} 0, & \text{if } q_i \in \{[CLS], [SEP]\} \text{ or } q_i, k_i \in SC \text{ or } \langle q_i, k_i \rangle \in E \cup E' \\ -\infty, & \text{otherwise} \end{cases} \tag{8}$$

$$\hat{X} = [head_1, \ldots, head_u]W_n^O \tag{9}$$

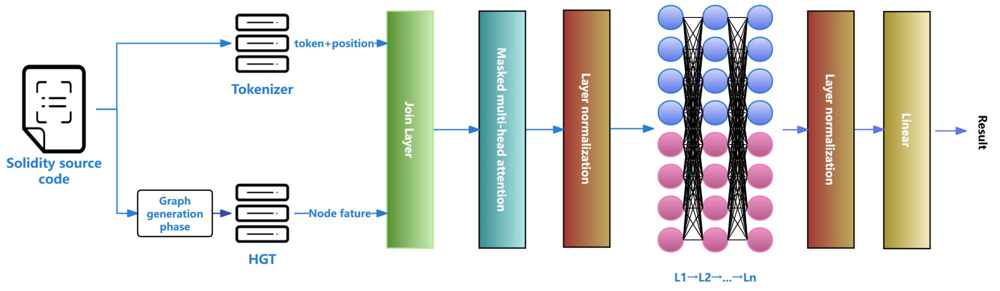


Figure 5. Vulnerability detection phase.

In Equation (6), $X^{(n-1)} \in \mathbb{R}^{|I| \times d_h}$ is the output of $(n-1)_{th}$ Transformer layers, X, W are vectors, and Q, K, V are triplets. $X^{(n-1)}$ is linearly projected onto the triplets of Q_i, K_i, and V_i using model parameters W_i^Q,W_i^K,$W_i^V \in \mathbb{R}^{d_h \times d_k}$. To incorporate a graph structure within the Transformer and illustrate dependencies between graph nodes, we adopt the approach of GraphCodeBERT [28], utilizing graph-guided masked attention to depict the interactions among tokens. Graph-guided masked attention is implemented through the mask matrix M. In Equation (7), *head* is the head in multi-head attention, d_k is the dimension of the *head*, M is the mask matrix, $M \in \mathbb{R}^{|I| \times |I|}$, where if the i_{th} token and the j_{th} token are associated, then $M_{ij} = 0$, otherwise it is $-\infty$. The calculation process of M is shown in Equation (8), $[CLS]$ is a special mark in front of the set, $[SEP]$ is a separator, $SC = \{sc, sc_2, \ldots, sc_n\}$ is the set of tokens in the source code, E is the set of edges $\{e_1, e_2, \ldots, e_l\}$ in the heterogeneous contract semantic graph HCG, representing the control flow and data flow relationship between nodes, and E' is a set indicating the association between the smart contract source code token and the nodes in the heterogeneous contract semantic graph (HCG). Attention computations are permissible between nodes v_i and v_j when they are directly connected (i.e., $\langle v_i, v_j \rangle \in E$) or are the same node (i.e., $i = j$). To represent the relationship between the source code tokens and the nodes in the heterogeneous contract semantic graph, we first define a set E'. If node v_i is determined by the token sc_j in the source code (i.e., $\langle v_i, \mathrm{sc}_j \rangle$, $\langle \mathrm{sc}_j, v_i \rangle \in E'$), they are allowed to perform attention computation with each other; in other cases, the attention is masked by assigning an attention score of $-\infty$. After the Softmax calculation in Equation (7), it is assigned a value of 0. In Equation (9), u is the number of heads in the multi-head attention, and $W_n^O \in \mathbb{R}^{d_h \times d_h}$ is the model parameter.

After the n_{th} layer of the Transformer model, layer normalization is employed for regularization. Subsequently, the output y, which represents the probability of the contract containing a vulnerability, is derived using a linear layer followed by a *Sigmoid* [35] function, as demonstrated in Equation (10). A loss function is then formulated to measure the discrepancy between this output y and the target value, where the target is set to 1 if the smart contract exhibits a specific vulnerability, and 0 otherwise. Finally, the backpropagation algorithm is utilized to train the network.

$$y = Sigmoid(X^n) \tag{10}$$

4. Experiment

In this section, we undertake a comprehensive empirical analysis of the method we propose, utilizing datasets that are publicly accessible. This evaluation is designed to rigorously test the effectiveness of our approach. To systematically assess the performance of our method, we have articulated several specific research questions:

- RQ1: Is our proposed method capable of effectively identifying the four most prevalent vulnerabilities in smart contracts, and does it outperform existing methods in this regard? We address this question by comparing the accuracy, precision, recall, and F1-score metrics.
- RQ2: What is the contribution of different modules in the proposed method to vulnerability detection? This question investigates the contribution of various modules to the model, including the heterogeneous contract semantic graph and the pre-trained model. We designed ablation experiments to answer this question.

4.1. Experimental Setup

Experiment environment: To conduct the experimental analysis, we utilized the open-source tool solc-typed-ast https://github.com/Consensys/solc-typed-ast, accessed on 11 March 2024, to parse Ethereum Solidity source code into a standard AST format. This was followed by further data optimization operations, such as removing comments. We developed a heterogeneous contract semantic graph generator based on the standardized AST information. The neural network was designed and implemented using the PyTorch framework. All experiments were performed on a physical machine running the Ubuntu 22.04 operating system, equipped with an Intel Xeon Silver 4310 processor with a base frequency of 2.1 GHz, 64 GB of RAM, and an NVIDIA A10 GPU with 24 GB of VRAM. Model training acceleration was achieved using the CUDA 12.4 computing library. The development environment included Visual Studio Code software (version 1.86), the PyTorch (version 2.2) framework, a Node.js (version 20.8.0) environment, and programming languages such as Python (version 3.11) and JavaScript (ECMAScript 2023).

Comparative Methods: To evaluate the effectiveness of the proposed method, we compared its performance with 12 state-of-the-art smart contract vulnerability detection methods. These include six open-source detection methods based on different neural network models: VanillaRNN [8], LSTM [36], GRU [36], DR-GCN [10], TMP [10], and CGE [22]. Additionally, we compared our method with six traditional smart contract vulnerability detection tools from top-tier conferences and journals: Oyente [17], Contractfuzzer [18], Mythril [11], Slither [7], Smartcheck [19], and Securify [37].

Dataset: To assess the effectiveness of the proposed method, we constructed our dataset based on the open-source dataset Smartbug [38]. Smartbug is one of the most widely used public datasets in smart contract vulnerability detection research, containing 47398 Ethereum smart contracts labeled based on static analysis tools. However, due to the errors in static analysis tools, this labeling method lacks complete accuracy. In order to conduct an accurate evaluation, we carefully selected those parts of Smartbug that have been manually verified and confirmed to be accurate vulnerability labels as our dataset in combination with relevant literature [16,30]. we first reclassified the data from several manually labeled smart contract datasets according to the four types of vulnerabilities we are studying. Then, we used automatic analysis tools to detect vulnerabilities in these contracts, ensuring that our dataset contained a certain number of contracts that traditional detection tools failed to classify as vulnerabilities. In addition, we removed comments from the code, excluded contracts with less than 100 lines of code, and deleted duplicate contracts, which had different addresses but the same main functions determined by code similarity analysis. This process produced the final dataset we used in our experiments. The dataset's variety and frequency of vulnerabilities are detailed in Table 1 . The dataset is randomly divided into three parts, each accounting for a different proportion: 70% for the training set, 20% for the validation set, and 10% for the test set. The neural network model uses the training set to learn the various vulnerability features in the smart contract, while the validation set is used to adjust parameters during model training to avoid model overfitting. The test set is used to evaluate the generalization ability of the model in vulnerability detection. Such a data allocation strategy helps to conduct a comprehensive and rigorous evaluation of the model.

Table 1. Vulnerability Counts Across Different Categories.

Reentrancy	Transaction State Dependency	Block Info Dependency	Nested Call	Safe
973	895	1030	863	1097

Parameter setting: The default parameters of the model are as follows: Batch size: 16, Initial learning rate: 2×10^{-5}, Dropout rate: 0.1, Optimizer: use AdamW, weight decay is set to 1×10^{-4}.

4.2. Evaluation Metrics

In the experiments presented in this paper, we employed four widely-used evaluation metrics to comprehensively assess the performance of the smart contract vulnerability prediction model: *Accuracy*, *Recall*, *Precision* and *F1-score*. The calculations for these metrics are based on the four crucial components in the confusion matrix: True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). The specific formulas for these calculations are detailed as follows:

$$Accuracy = 100 \times \frac{TP + TN}{TP + FP + TN + FN} \tag{11}$$

$$Recall = 100 \times \frac{TP}{TP + FN} \tag{12}$$

$$Precision = 100 \times \frac{TP}{TP + FP} \tag{13}$$

$$F1 = 200 \times \frac{Precision \times Recall}{Precision + Recall} \tag{14}$$

Note: All calculated values in the article are multiplied by 100 and expressed as percentages.

4.3. Ablation Experiment

This section aims to validate the individual contributions of each component in the proposed method. First, to verify the contribution of pre-training techniques, we normalized the parameters of the pre-trained model (GraphCodeBERT [28]) and trained the model using the same input. For the smart contract files in the dataset, the canonical abstract syntax tree (AST) is first parsed by the solc-typed-ast tool. Based on the AST, a heterogeneous contract semantic graph is constructed, and features are extracted through the HGT network. Before training, GraphCodeBERT's parameters are normalized to eliminate the pre-training advantage. We denote this variant as $method - WP$. During the training phase, the loss rate decreased slowly. After a period of training, we tested the results on the test set, as shown in Table 2. The $method - WP$ exhibited a substantial decline in performance across various types of vulnerability detection tasks. This outcome suggests that pre-training techniques can boost the generality of models, lessen the learning burden, and serve an indispensable role in the detection of vulnerabilities.

Table 2. Performance comparison of our method and its variants on four vulnerability detection tasks.

Methods	Reentrancy				Block Info Dependency				Transaction State Dependency				Nested Loop			
	Acc	Rec	Pre	F1	Acc	Rec	Pre	F1	Acc	Rec	Pre	F1	Acc	Rec	Pre	F1
Method-WP	55.32	34.68	48.03	59.1	44.69	50.52	47.92	49.19	41.24	47.96	31.74	51.24	39.56	36.28	32.09	34.06
Method-HOG	80.09	81.18	72.15	76.4	81.3	80.68	78.42	79.53	74.11	76.39	68.92	72.46	70.23	65.08	71.44	68.11
Our method	90.96	91.62	89.16	90.37	89.57	87.62	91.43	89.50	88.46	86.20	90.37	88.26	87.34	84.91	89.77	87.30

To study the contribution of the heterogeneous contract semantic graph, we compared the performance differences between homogeneous and heterogeneous graph analyses. For the smart contract files in the dataset, the canonical abstract syntax tree (AST) is first parsed by the solc-typed-ast tool. In the contract graph construction phase, the heterogeneous contract semantic graph is replaced by the isomorphic graph generated by the method proposed by Liu [22], so this variant is recorded as $method - HOG$. Then, the graph embedding is completed by GAT (graph attention neural network), and finally the obtained graph features and source code are used as input to train the GraphCodeBERT model loaded with pre-trained parameters. After training, the results tested on the test set are shown in Table 2. We observed that the original architecture of our method significantly outperformed method-HOG in all aspects. This result suggests that the heterogeneous contract semantic graph contains finer-grained semantic information in its graphical features, which is crucial for the smart contract vulnerability detection task.

4.4. Comparison with Other Deep Learning Based Methods

The experiments compared the proposed method with 12 state-of-the-art methods, including six open-source detection methods based on different neural network models: VanillaRNN [8], LSTM [36], GRU [36], DR-GCN [10], TMP [10], and CGE [22]. Table 3 presents the *Accuracy*, *Recall*, *Precision* and F1-score values of our method compared to the six detection methods based on neural network models.

Table 3. Performance comparison with neural network-based methods.

Methods	Reentrancy				Block Info Dependency				Transaction State Dependency				Nested Call			
	Acc	Rec	Pre	F1	Acc	Rec	Pre	F1	Acc	Rec	Pre	F1	Acc	Rec	Pre	F1
Vanilla-RNN	50.12	55.84	46.33	50.72	49.77	44.59	51.17	52.15	52.43	50.20	54.65	52.40	51.98	50.80	48.76	49.71
LSTM	53.29	59.97	48.45	53.79	50.79	57.23	51.42	54.17	53.15	60.24	51.86	55.74	55.07	56.35	50.83	53.43
GRU	57.80	72.17	52.18	60.68	62.76	59.85	65.43	62.61	62.76	59.85	65.43	62.61	59.65	57.43	61.98	59.97
DR-GCN	78.96	77.05	73.24	75.09	75.53	72.89	78.11	75.51	-	-	-	-	-	-	-	-
TMP	82.15	80.90	80.39	80.64	80.87	78.39	83.45	80.88	-	-	-	-	-	-	-	-
CGE	85.32	81.25	87.46	84.23	86.21	84.10	85.96	85.03	-	-	-	-	-	-	-	-
Our method	90.96	91.62	89.16	90.37	89.57	87.62	91.43	89.50	88.46	86.20	90.37	88.26	87.34	84.91	89.77	87.30

"-" means the corresponding tool does not support detecting this type of vulnerability.

The initial three models utilize text sequences from smart contract codes as inputs, while the subsequent three models are based on the graph structure data of smart contracts. These models represent typical approaches in the field of smart contract vulnerability detection and have been widely adopted as benchmark methods in recent studies [10,30,37]. Considering that none of the compared methods support other block information dependency vulnerabilities except timestamp dependency, in order to ensure the fairness of the comparison, we limit the scope of method evaluation and only verify and compare the detection effect of timestamp dependency vulnerabilities. It can be seen that in smart contract reentrancy vulnerability detection, our method improves the accuracy by 52.11% compared to the best-performing text sequence modeling model, GRU, and by 6.61% compared to the best-performing graph structure modeling model, CGE. In detecting block information dependency vulnerabilities in smart contracts, our method improves the accuracy by 42.71% compared to the best-performing text sequence modeling model, GRU, and by 4.90% compared to the best-performing graph structure modeling model, CGE. We attribute this improvement to our proposed heterogeneous contract semanti graph, which contains richer semantic information and higher separation between different features compared to homogeneous graphs. Methods based on graph structure data modeling tend to outperform those based on text sequence modeling, which also proves that code is not merely a sequence of words, and ignoring structural information such as data flow and control flow can degrade performance. For the detection of transaction state dependency vulnerabilities and Nested Call vulnerabilities, as the three graph structure-based methods do not support these types, we only compare with the text sequence-based methods. The results show that compared to the best-performing GRU, our method improves the accuracy by 40.94% and 46.42%, respectively.

4.5. Compared with Traditional Tools

In addition to comparing our method with neural network-based approaches, we also compared it with state-of-the-art traditional smart contract vulnerability detection tools. Based on the detection targets and main features of these tools [39], we used Smartcheck [19], Securify [37], and Slither [7] for source code-level vulnerability detection, and Oyente [17], Contractfuzzer [18], and Mythril [11] for bytecode-level vulnerability detection. In order to meet the input requirements of Oyente, Contractfuzzer and Mythril tools, we use the compiler to generate corresponding bytecodes for the smart contracts in the dataset for them to use. The results are shown in Table 4.

Table 4. Performance comparison with traditional tools.

Methods	**Reentrancy**				**Block Info Dependency**				**Transaction State Dependency**				**Nested Call**			
	Acc	**Rec**	**Pre**	**F1**	**Acc**	**Rec**	**Pre**	**F1**	**Acc**	**Rec**	**Pre**	**F1**	**Acc**	**Rec**	**Pre**	**F1**
Smartcheck	40.43	41.25	39.11	40.15	52.91	39.70	51.17	52.15	30.66	33.89	40.67	36.97	47.26	50.12	55.99	52.99
Oyente	60.23	62.35	65.12	63.46	39.45	41.23	43.56	42.37	-	-	-	-	42.78	39.75	46.89	43.03
Contractfuzzer	67.54	61.44	49.26	54.68	63.99	65.68	67.89	66.75	-	-	-	-	-	-	-	-
Mythril	65.35	69.91	50.08	58.36	70.23	63.31	69.88	66.43	77.12	51.32	59.70	55.19	60.23	55.71	60.38	55.71
Securify	73.88	75.79	68.45	71.93	-	-	-	-	-	-	-	-	-	-	-	-
Slither	80.35	87.10	82.57	84.77	77.12	74.28	68.42	71.23	60.96	82.09	62.07	71.27	55.37	27.69	61.91	38.27
Our method	90.96	91.62	89.16	90.37	89.57	87.62	91.43	89.50	88.46	86.20	90.37	88.26	87.34	84.91	89.77	87.30

"-" means the corresponding tool does not support detecting this type of vulnerability.

It can be observed that traditional vulnerability detection tools do not perform well. The best-performing traditional tool, Slither, achieved an accuracy of 80.35% in reentrancy vulnerability detection, while our method significantly improved the accuracy by 13.20%. We believe the poor performance of traditional methods is due to their heavy reliance on simple and fixed vulnerability detection patterns. For example, Mythril uses symbolic execution, mixed concrete execution, and control flow analysis to detect vulnerabilities in smart contracts. In reentrancy vulnerability detection, Mythril determines if a contract has a reentrancy vulnerability based on whether there are internal function calls or state variable modifications following a low-level call to *call.value*. Additionally, tools that use bytecode as input lack source code semantic information, which can lead to decreased detection performance. For block information dependency vulnerability detection, since Securify does not support the detection of such vulnerabilities, we only compared our method to the other five methods. Our method outperforms traditional methods on all four metrics and achieves 16.1% accuracy improvement over the best traditional method, Slither. The poor performance of some traditional tools may be attributed to their simplistic approach of checking for the presence of block information statements like *block.timestamp* to determine the existence of timestamp dependency vulnerabilities. Since only Mythril, Slither, and Smartcheck support the detection of transaction state dependency vulnerabilities and Nested call vulnerabilities, we compared our method solely with these three tools. The results show that our method improved accuracy by 14.7% and 26% in detecting transaction state dependency vulnerabilities and Nested call vulnerabilities, respectively, compared to the best-performing traditional tool, Mythril.

4.6. Comparison of Runtime Resource Consumption

To verify the usability of the proposed method, we evaluated the average time and memory consumption of the proposed method and other baseline methods in detecting contracts in the dataset. Since only reentrancy vulnerabilities can be detected by all methods, to ensure the fairness of the evaluation and the accuracy of the results, we screened out contracts containing reentrancy vulnerabilities and security contracts from the original dataset to form a sub-dataset, and shut down all background processes for evaluation in a clean experimental environment.

Traditional tools are primarily based on techniques such as symbolic execution, static analysis, or fuzz testing. These methods have most of their computational demands concentrated on the CPU and incur memory overhead during both static analysis and dynamic execution. Therefore, we measured their memory consumption. For deep learning-based methods, the VRAM consumption during inference is more indicative of model complexity and resource utilization. Thus, we primarily measured their VRAM consumption during the inference process. To ensure the reliability of the results, we ran each experiment multiple times in the clean environment and averaged the results to obtain the final evaluation. The results are shown in Table 5.

Table 5. Running time and memory consumption of different methods.

Methods	Acc	Recall	Precision	F1	Avg Time (s)	Memory (MB)	VRAM (MB)
Smartcheck	40.18	41.36	39.15	40.11	0.92	**3.58**	-
Oyente	60.05	62.77	65.21	63.41	2.50	174.44	-
Contractfuzzer	68.07	60.83	48.96	54.80	1.76	85.35	-
Mythril	65.13	69.36	49.83	57.95	2.39	161.40	-
Securify	73.38	75.86	68.52	71.50	2.73	170.59	-
Slither	81.15	87.86	82.00	84.71	0.89	4.27	-
Vanilla-RNN	50.46	55.78	46.40	50.24	**0.35**	-	71.34
LSTM	52.78	60.09	48.14	54.06	0.44	-	125.39
GRU	58.16	72.09	51.81	60.69	0.39	-	98.35
DR-GCN	79.24	77.17	73.05	74.96	0.65	-	217.53
TMP	81.54	80.61	79.66	80.08	0.73	-	393.75
CGE	85.56	81.19	88.32	83.58	0.91	-	674.20
Our method	90.96	91.62	89.16	90.37	0.84	-	650.83

Among traditional tools, the static analysis-based method Slither has the fastest execution speed and consumes the least memory, because static analysis usually only needs to traverse the code once, generate an abstract syntax tree or control flow graph, and use predefined rules for analysis. The symbolic execution-based method Securify generates a large number of symbolic paths and symbolic expressions during execution. For multiple conditional branches in complex code, the symbolic path may grow exponentially, resulting in a large amount of memory consumption. The execution time and memory consumption of deep learning-based methods are highly correlated with model complexity. Although the three methods using text sequences as input have faster inference speeds, they exhibit significant deficiencies in terms of accuracy. In contrast, methods based on graph structure modeling, while having increased inference time, significantly improve detection accuracy by better capturing the structural information within smart contracts. Given the substantial economic value associated with smart contracts, we believe that increasing computational resource consumption to enhance detection accuracy is both reasonable and worthwhile.

Compared to Slither, the best-performing traditional method, our approach achieves a 13% improvement in accuracy with similar execution time, greatly reducing the likelihood of missing potential vulnerabilities. Additionally, compared to CGE, the best neural network-based model, our method achieves a 6% increase in detection accuracy while maintaining comparable inference time and VRAM consumption. This demonstrates that our method not only improves performance while keeping resource consumption stable but also more effectively leverages model complexity to enhance the detection of contract vulnerabilities. Our method can provide higher security in actual smart contract audits, especially for high-value smart contracts. This performance improvement can effectively reduce the risk of economic losses caused by vulnerabilities. In addition, maintaining a balance between reasoning efficiency and resource consumption also makes our method more practical in large-scale or batch contract audit scenarios.

5. Conclusions

In this paper, we propose a smart contract vulnerability detection method based on pre-training technology and heterogeneous contract semantic graphs. Compared with existing methods, the heterogeneous contract semantic graph proposed in this paper better captures the dependency between program variables and functions. The graph embedding is achieved through the heterogeneous graph neural network HGT, and the introduction of pre-training technology enables efficient detection of smart contract vulnerabilities. Experimental results show that the proposed method performs well in detecting four of the most severe and widespread vulnerabilities. This approach not only covers a broader range of vulnerability types but also achieves higher detection accuracy compared to existing methods. It serves as an effective tool for the preliminary screening of vulnerabilities in smart contracts, significantly improving the efficiency with which developers identify potential issues. The method holds broad application prospects in the field.

However, our method has certain limitations:

- The scope of our method is currently limited to Ethereum and Solidity, and it does not support the detection of vulnerabilities in smart contracts written in other languages (such as Vyper, Rust, and Go).
- The dataset used in our experiments was manually curated based on existing research, and this process may introduce some degree of subjectivity, potentially leading to false positives.
- This study focuses on verifying the effectiveness of our approach using only four specific vulnerability types. As blockchain technology evolves and programming languages continue to advance, the types of vulnerabilities will also diversify.

6. Future Work

In future work, we aim to extend the capabilities of our method in several key areas. First, we plan to broaden the scope of our approach to support vulnerability detection in smart contracts written in other blockchain programming languages, such as Vyper, Rust, and Go. This will enable our method to be applied across a wider range of platforms and ecosystems beyond Ethereum and Solidity, making it more versatile and robust for developers working with different technologies. Second, we plan to extend our evaluation beyond the four specific vulnerability types currently used. As blockchain technology evolves, new types of vulnerabilities will emerge, and it is crucial to ensure that our method can detect a broader array of vulnerabilities. This includes incorporating emerging vulnerability types such as Gas optimization issues, flash loan attacks, etc. Furthermore, In addition, our current approach focuses on detecting vulnerabilities in a single smart contract. As blockchain technology continues to develop, decentralized applications (DApps) are becoming a mainstream use case for blockchain systems. Therefore, developing methods for detecting vulnerabilities between multiple interconnected contracts and components in DApps will become a valuable and important area for future research. In the future, we will explore the possibility of applying the current approach to DApp vulnerability detection.

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

Funding: This research received no external funding.

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 conflicts of interest.

Appendix A

Appendix A provides detailed descriptions and code examples of four smart contract vulnerabilities, as well as the solc-typed-ast node type table to help readers better understand the technical details mentioned in this article.

Appendix A.1. Reentrancy

One crucial feature of smart contracts is their ability to call and utilize code from other external contracts. However, these external calls can potentially be hijacked by attackers who use fallback functions to execute additional code, including calls that return to the contract's own code. In Solidity, the fallback function is a special, unnamed function that takes no parameters and does not return a value. It is called when no matching function signature is found, or when the contract directly receives Ether. Thus, attackers can construct a contract at an external address that includes malicious code in the fallback function. When a contract calls certain functions at this address, it triggers the malicious code [40]. The term "reentrancy" refers to the scenario where an external malicious contract uses function calls to "re-enter" and manipulate the execution process of the vulnerable contract.

Reentrancy Example Example: Listing A1 illustrates a contract with a reentrancy vulnerability, the ReentrantBank contract, which acts as a bank holding users' Ether. Users can withdraw Ether through the *withdraw()* function, which contains the reentrancy vulnerability. The attacker's contract exploits this vulnerability to steal funds from the bank contract. In step 1, the attacker calls the *withdraw()* function in the ReentrantBank contract to withdraw Ether. In step 2, the ReentrantBank contract sends Ether to the attacker by executing *sender.call{value: _amount}("")*. In step 3, instead of proceeding to the next expected step—deducting the attacker's balance—the process enters the *fallback()* function of the Attacker contract because the Ether transfer is conducted using *sender.call{value: _amount}("")*. In step 4, within the *fallback()* function, the attacker calls the *withdraw()* function again to extract Ether. This cycle continues repeatedly until the Ether in the ReentrantBank contract is depleted.

Listing A1. Reentrancy Example.

```
// Bank contract vulnerable to reentrancy attack
contract ReentrantBank {
mapping(address => uint256) public balances;  // Stores each user's balance
function deposit() public payable {
require(msg.value > 0);
balances[msg.sender] += msg.value;  // Update the user's balance
}
function withdraw(uint256 _amount) public {
require(balances[msg.sender] >= _amount);  // Ensure the user has enough balance
// Vulnerability: Sends Ether to the caller before updating the balance
(bool sent, ) = msg.sender.call{value: _amount}("");  // Call to external contract
require(sent, "Failed to send Ether");
balances[msg.sender] -= _amount;  // Update the user's balance (should be done before the call)
}
function getBalance() public view returns (uint256) {
return address(this).balance;
}
}
// Attacker contract exploiting the reentrancy vulnerability
contract Attacker {
ReentrantBank public bank;
uint256 public amount;
constructor(address _bankAddress) {
bank = ReentrantBank(_bankAddress);
}
fallback() external payable {
if (address(bank).balance >= amount) {
bank.withdraw(amount);  // Recursively call withdraw to drain funds
}
}
function attack(uint256 _amount) external payable {
require(msg.value >= _amount);
amount = _amount;
bank.deposit{value: msg.value}();  // Deposit Ether into the bank contract
bank.withdraw(_amount);  // Trigger the withdraw function, starting the reentrancy attack
}
}
```

Appendix A.2. Transaction State Dependency

In Solidity, there is a global variable *tx.origin* that traces back through the entire call stack to return the address of the account that originally initiated the call (or transaction). If this variable is used within a smart contract to check if the caller has the correct permissions for sensitive functions, it can lead to severe consequences. This is because tx.origin can be manipulated in a way that deceives the contract into identifying the call as originating from a trusted source when, in fact, it might be coming from an attacker. This vulnerability can be exploited to grant unauthorized access to critical functions that should be protected, leading to potential losses or other security breaches in the contract [14].

Transaction State Dependency Example: As shown in Listing A2, attackers can bypass permission checks by exploiting the logic on line 15. By using this method, anyone can successfully execute the *withdraw()* function on line 6 to withdraw Ether from the contract.

Listing A2. Transaction State Dependency.

```
contract Vulnerable {
address public owner;  // Stores the contract owner's address
constructor() {
owner = msg.sender;  // msg.sender is the address that deployed the contract
}
function withdraw() public {
// Vulnerability: Using tx.origin instead of msg.sender allows an attack through an intermediary contract
require(tx.origin == owner, "Only owner can withdraw");  // Check if the original transaction initiator is the owner
payable(owner).transfer(address(this).balance);
}
receive() external payable {}
}
contract Attacker {
Vulnerable public vulnerableContract;
constructor(address _vulnerableAddress) {
vulnerableContract = Vulnerable(payable(_vulnerableAddress));
}
function attack() public {
// When this function is called by an attacker, tx.origin refers to the attacker,
// but msg.sender in Vulnerable contract will be this contract, bypassing the ownership check
vulnerableContract.withdraw();
}
receive() external payable {}
}
```

Appendix A.3. Block Info Dependency

Smart contracts can access block information (such as *block.timestamp*, *block.number*, and *block.hash*) as part of their execution context. However, relying on these blockchain environmental variables to determine their execution logic can lead to vulnerabilities related to block information dependency [14]. For instance, using attributes of future blocks as seeds for generating random numbers to determine the winners in a lottery game can be problematic.

Block Info Dependency Example: In Listing A3, a lottery game contract is shown, which selects a winner using the *random()* function (line 6). The *random()* function generates a random number by performing calculations based on *block.timestamp* (line 11). When the game accumulates a significant amount of Ether, miners are incentivized to manipulate these values within certain limits to increase their chances of winning.

Listing A3. Block Info Dependency.

```
contract Lottery {
address public owner;
address[] public players;
function pickWinner() public onlyOwner {
require(players.length > 0, "No players participated");
uint index = random() % players.length;  // Select a winner based on the random function
// Transfer the contract balance to the selected winner
payable(players[index]).transfer(address(this).balance);
players = new address;
}
// Function to generate a pseudo-random number, using block information (timestamp and difficulty)
// Vulnerability: Block information can be influenced by miners, making the random number predictable or manipulable
function random() private view returns (uint) {
// Generate a pseudo-random number using block.timestamp, block.difficulty, and~players array
return uint(keccak256(abi.encodePacked(block.timestamp, block.difficulty, players)));
}
receive() external payable {}
}
```

Appendix A.4. Nested Call

In the Ethereum execution environment, using the *CALL* instruction in a contract requires paying gas, with a basic cost of 700 gas. If the call involves transferring a non-zero value, an additional 9000 gas is required [14]. If a loop contains a *CALL* operation but does not restrict the number of iterations, there is a risk that the gas cost could exceed the limit. In such cases, the transaction will be terminated and reverted (rolled back), but the gas already consumed will not be refunded. This could lead to users or contract owners incurring high costs without achieving any results.

Nested Call Example: In Listing A4, an attacker can maliciously increase the number of iterations in a loop, Each call to the *send()* function during each iteration will use the CALL instruction to switch context in the EVM, which will consume a lot of gas. When the gas consumption exceeds the limit, the transaction will be reverted, and only unused gas will be refunded. The gas already consumed during execution will not be refunded, causing the user or contract holder to pay high fees without any results.

Listing A4. Nested Call.

```
contract VulnerableContract {
function callExternal() public payable {
for (uint i = 0; i<member.length; i++) {
member[i].send(1 wei);}
}
}
```

Appendix A.5. Solc-Typed-ast Node Type Table

Table A1. Solc-typed-ast node type table.

Functional Category	**Node Type**
Structural and Operation	ContractDefinition, EnumDefinition, ErrorDefinition, EventDefinition, FunctionDefinition, ModifierDefinition, SourceUnit, Block, Break, Continue, DoWhileStatement, EmitStatement, ExpressionStatement, ForStatement, IfStatement, InlineAssembly, PlaceholderStatement, PragmaDirective, Return, RevertStatement, Throw, TryCatchClause, TryStatement, UncheckedBlock, VariableDeclarationStatement, WhileStatement, NewExpression, Assignment, BinaryOperation, Conditional, TupleExpression, UnaryOperation, IndexAccess, IndexRangeAccess, MemberAccess
Variables and calling	FunctionCall, Identifier, IdentifierPath, UserDefinedValueTypeDefinition, VariableDeclaration, ModifierInvocation, ParameterList, StructDefinition
other	ImportDirective, PragmaDirective, OverrideSpecifier, StructuredDocumentation, UsingForDirective, InheritanceSpecifier, ArrayTypeName, ElementaryTypeName, EnumValue, FunctionTypeName, UserDefinedTypeName, FunctionCallOptions, Literal

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://www.ussc.gov/sites/default/files/pdf/training/annual-national-training-seminar/2018/Emerging_Tech_Bitcoin_Crypto.pdf (accessed on 1 July 2024). [CrossRef]
2. Shao, Q.; Jin, C.; Zhang, Z.; Qian, W.; Zhou, A. Blockchain: Architecture and Research Progress. *Chin. J. Comput.* **2018**, *41*, 969–988. (In Chinese with English abstract) [CrossRef]
3. SlowMist. 2023 Blockchain Security and Anti-Money Laundering Annual Report. 2024. Available online: https://www.slowmist.com/report/2023-Blockchain-Security-and-AML-Annual-Report(EN).pdf (accessed on 11 February 2024).
4. Mossberg, M.; Manzano, F.; Hennenfent, E.; Groce, A.; Grieco, G.; Feist, J.; Brunson, T.; Dinaburg, A. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1186–1189. [CrossRef]
5. So, S.; Hong, S.; Oh, H. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts Through Language Model-Guided Symbolic Execution. In Proceedings of the 30th USENIX Security Symposium, Online, 11–12 August 2021; pp. 1361–1378.
6. Torres, C.; Iannillo, A.; Gervais, A.; State, R. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In Proceedings of the 2021 IEEE European Symposium on Security and Privacy (EuroS&P), Vienna, Austria, 6–10 September 2021; pp. 103–119. [CrossRef]
7. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In Proceedings of the 2nd IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, 27 May 2019; pp. 8–15. [CrossRef]
8. Goller, C.; Kuchler, A. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In Proceedings of the International Conference on Neural Networks (ICNN '96), Washington, DC, USA, 3–6 June 1996; pp. 347–352.
9. Sak, H.; Senior, A.; Beaufays, F. Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. In Proceedings of the Fifteenth Annual Conference of the International Speech Communication Association, Perth, WA, Australia, 27 November–1 December 2014.
10. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart Contract Vulnerability Detection using Graph Neural Network. In Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI), Yokohama, Japan, 11–17 July 2020; pp. 3283–3290.
11. Mueller, B. Mythril-Reversing and Bug Hunting Framework for the Ethereum Blockchain. 2017. Available online: https://pypi.org/project/mythril/0.8.2 (accessed on 11 February 2024).
12. di Angelo, M.; Salzer, G. Consolidation of Ground Truth Sets for Weakness Detection in Smart Contracts. *arXiv* **2023**, arXiv:2304.11624.
13. SWC. Smart Contract Weakness Classification. 2023. Available online: https://swcregistry.io/ (accessed on 11 February 2024).
14. Chen, J.; Xia, X.; Lo, D.; Grundy, J.; Luo, X.; Chen, T. Defining Smart Contract Defects on Ethereum. *IEEE Trans. Softw. Eng.* **2022**, *48*, 327–345. [CrossRef]
15. Wang, W.; Song, J.; Xu, G.; Li, Y.; Wang, H.; Su, C. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Trans. Netw. Sci. Eng.* **2021**, *8*, 1133–1144. [CrossRef]
16. Luo, F.; Luo, R.; Chen, T.; Qiao, A.; He, Z.; Song, S.; Jiang, Y.; Li, S. SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), Lisbon, Portugal, 12–24 April 2024. [CrossRef]
17. Luu, L.; Chu, D.; Olickel, H.; Saxena, P.; Hobor, A. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, 24–28 October 2016; pp. 254–269. [CrossRef]

18. Jiang, B.; Liu, Y.; Chan, W. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 259–269. [CrossRef]
19. Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. SmartCheck: Static Analysis of Ethereum Smart Contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Gothenburg, Sweden, 27 May–3 June 2018; pp. 9–16. [CrossRef]
20. Hildenbrandt, E.; Saxena, M.; Rodrigues, N.; Zhu, X.; Daian, P.; Guth, D.; Moore, B.; Park, D.; Zhang, Y.; Stefanescu, A.; et al. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF), Oxford, UK, 9–12 July 2018; pp. 204–217. [CrossRef]
21. Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. ZEUS: Analyzing Safety of Smart Contracts. In Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 18–21 February 2018; pp. 1–12. [CrossRef]
22. Liu, Z.; Qian, P.; Wang, X.; Zhuang, Y.; Qiu, L.; Wang, X. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Trans. Knowl. Data Eng.* **2023**, *35*, 1296–1310. [CrossRef]
23. Devlin, J.; Chang, M.; Lee, K.; Toutanova, K. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Minneapolis, MN, USA, 2–7 June 2019; pp. 4171–4186. [CrossRef]
24. Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. Improving Language Understanding by Generative Pre-Training. 2018. Available online: https://paperswithcode.com/paper/improving-language-understanding-by (accessed on 11 February 2024).
25. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the 2020 Findings of the Association for Computational Linguistics (EMNLP), Online Event, 16–20 November 2020; pp. 1536–1547. [CrossRef]
26. Wang, Y.; Wang, W.; Joty, S.; Hoi, S. Codet5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP), Punta Cana, Dominican Republic, 7–11 November 2021; pp. 8696–8708.
27. Buratti, L.; Pujar, S.; Bornea, M.; McCarley, S.; Zheng, Y.; Rossiello, G.; Morari, A.; Laredo, J.; Thost, V.; Zhuang, Y.; et al. Exploring Software Naturalness through Neural Language Models. *arXiv* **2020**, arXiv:2006.12641.
28. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-Training Code Representations with Data Flow. In Proceedings of the 9th International Conference on Learning Representations (ICLR), Virtual Event, Austria, 3–7 May 2021.
29. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. In Proceedings of the 6th International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
30. Wu, H.; Zhang, Z.; Wang, S.; Lei, Y.; Lin, B.; Qin, Y.; Zhang, H.; Mao, X. Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-Training Techniques. In Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), Wuhan, China, 25–28 October 2021; pp. 378–389.
31. Dong, Y.; Chawla, N.; Swami, A. metapath2vec: Scalable Representation Learning for Heterogeneous Networks. *Acm Trans. Knowl. Discov. Data* **2017**. [CrossRef]
32. Wang, X.; Ji, H.; Shi, C.; Wang, B.; Ye, Y.; Cui, P.; Yu, P. Heterogeneous Graph Attention Network. In Proceedings of the The World Wide Web Conference (WWW '19), Raleigh, NC, USA, 26–30 April 2019; pp. 2022–2032. [CrossRef]
33. Hu, Z.; Dong, Y.; Wang, K.; Sun, Y. Heterogeneous Graph Transformer. In Proceedings of the Web Conference 2020 (WWW '20), Taipei, Taiwan, 20–24 April 2020; pp. 2704–2710. [CrossRef]
34. Ethereum. Units and Globally Available Variables. 2023. Available online: https://docs.soliditylang.org/zh/latest/units-and-global-variables.html (accessed on 11 February 2024).
35. Zhang, Z.; Lei, Y.; Mao, X.; Li, P. CNN-FL: An Effective Approach for Localizing Faults Using Convolutional Neural Networks. In Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 445–455. [CrossRef]
36. Chung, J.; Gulcehre, C.; Cho, K.; Bengio, Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv* **2014**, arXiv:1412.3555.
37. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Buenzli, F.; Vechev, M. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, 15–19 October 2018; pp. 67–82.
38. Durieux, T.; Ferreira, J.; Abreu, R.; Cruz, P. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE), Seoul, Republic of Korea, 27 June–19 July 2020; pp. 530–541. [CrossRef]
39. Cui, Z.; Yang, H.; Chen, X.; Wang, L.Z. Progress in Smart Contract Security Vulnerability Detection. *J. Softw.* **2024**, *35*, 2235–2267. [CrossRef]
40. Antonopoulos, A.; Wood, G. *Mastering Ethereum: Building Smart Contracts and DApps*; O'Reilly Media: Sebastopol, CA, USA, 2018; p. 177.