Cybersecurity

RESEARCH

Open Access

MVD-HG: multigranularity smart contract vulnerability detection method based on heterogeneous graphs

Jingjie Xu[1], Ting Wang[1], Mingqi Lv[1], Tieming Chen[1*], Tiantian Zhu[1] and Baiyang Ji[1]

Abstract

Smart contracts have significant losses due to various types of vulnerabilities. However, traditional vulnerability detection methods rely extensively on expert rules, resulting in low detection accuracy and poor adaptability to novel attacks. To address these problems, in this paper, deep learning methods are combined with smart contract vulnerability code detection approaches. Abstract syntax trees (ASTs), which are special isomorphic graph structures, are an important bridge between source code and graph neural networks. By learning the AST, the model can understand the semantics of the source code. Moreover, graph neural networks have an increasing ability to address complex heterogeneous graphs. Therefore, control flow graphs are fused with data flow graphs on the basis of the ASTs to build heterogeneous graphs with richer code semantics. Furthermore, multigranularity analysis of the vulnerability detection results is performed, including coarse-grained contract-level vulnerability detection and fine-grained line-level vulnerability detection. Through this multigranularity detection approach, vulnerabilities in contracts can be identified and analysed more comprehensively, providing a richer perspective and more solutions for vulnerability detection. The experimental results show that the proposed multigranularity vulnerability detection method based on heterogeneous graphs (MVD-HG) improves both the accuracy and range of the detected vulnerability types in contract-level vulnerability detection tasks; moreover, in the line-level vulnerability detection task, the MVD-HG model achieves significant results and addresses the shortcomings of existing methods. In addition, based on code generation methods used in related fields, a data enhancement method based on the source code is developed, which effectively expands the experimental dataset to address the reduced credibility of the results due to insufficient amounts of data.

Keywords Smart contracts, Abstract syntax trees, Heterogeneous graphs, Vulnerability detection, Data augmentation

Introduction

In recent years, with the advancement of blockchain technology, smart contracts have received much attention from researchers in academia and industry. Smart contracts were originally designed to automatically execute predefined code in scenarios with no trusted third party. However, because smart contracts unconditionally follow predefined programs and record results on blockchain platforms that are difficult to tamper with, many malicious, irreversible events occur in the blockchain. The occurrence of malevolent events not only severely reduces the security of user property but also impacts the reliability of blockchain platforms. The DAO incident in 2016 and the Second Parity MultiSig Wallet incident in 2017 (Samreen and Alalfi 2021) resulted in losses of 3.6 million ether and 150,000 ether, respectively. Thus, smart contracts with vulnerabilities in their design attract many

*Correspondence:
Tieming Chen
tmchen@zjut.edu.cn
[1] College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310000, Zhejiang, China

© The Author(s) 2024. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

malicious users, demonstrating the importance of smart contract vulnerability detection.

To address the problem of vulnerable code in smart contracts, researchers have investigated various detection methods for identifying code vulnerabilities. Traditional methods for smart contract vulnerability analysis include static analysis, dynamic analysis, and formal verification methods (Tang et al. 2021). Static analysis methods include Oyente (Luu et al. 2016), Securify (Tsankov et al. 2018), SmartCheck (Tikhomirov et al. 2018), EtherTrust (Grishchenko et al. 2018), and SmartConDetect (Jeon et al. 2021); dynamic analysis methods include Contract-Fuzzer (Jiang et al. 2018), ReGuard (Liu et al. 2018), and Maian (Nikolić et al. 2018); and formal validation methods include ZEUS (Kalra et al. 2018) and KEVM (Hildenbrandt et al. 2018). These traditional analysis methods usually use manually defined expert rules for vulnerability detection. However, when new types of vulnerabilities appear, the rules need to be updated to identify these new vulnerabilities. With the widespread use of smart contracts and their changing application scenarios, traditional analysis methods based on manually defined expert rules have faced great challenges.

Compared with traditional methods, deep learning-based approaches can address the drawback of requiring experts to define vulnerability patterns and instead use neural networks to identify patterns. For example, MANDO was proposed by Nguyen et al. (2022), Peculiar was proposed by Wu et al. (2021), DR-GCN with TMP was proposed by Zhuang et al. (2021), and VDDL was proposed by Jiang et al. (2022).

However, existing deep learning-based methods for detecting vulnerabilities in smart contracts suffer from several shortcomings.

First, since most methods are based on graph neural networks, the program first needs to be represented as a graph structure. Most of the related studies have used specific types of graph structures; for example, Peculiar used CDFG, DR-GCN used Contract Graph, and MANDO used a heterogeneous graph consisting of CGs and CFGs. However, some of the semantic information may be lost when these graph structures are used. Moreover, they can capture information about only a few kinds of vulnerabilities, and their generalizability is relatively poor. For example, MANDO only considers control flow information and ignores the information available in data flows, which prevents effective detection if the vulnerability is caused by the data flow.

Second, most deep learning-based methods can perform only contract-level code vulnerability detection. The only method that can obtain more fine-grained line-level vulnerability detection results is MANDO. However, since MANDO is a metapath-based task, the researcher must manually specify all the metapaths, which relies considerably on the researcher's domain knowledge and complicates makes metapath design process. In addition, for existing attack types, these metapath rules can be bypassed by changing the attack steps without affecting the attack effect, while for new types of attacks, the corresponding metapath rule base needs to be updated, and existing methods have a low level of automation.

Third, in contrast to traditional methods, deep learning-based approaches require a large amount of data to train the model. However, it is difficult for researchers that are unfamiliar with smart contract development to perform meticulous data annotations, which makes it difficult and extremely laborious to build datasets. Therefore, it is necessary to hire professionals to manually annotate the data or use traditional automated vulnerability detection methods to reduce the annotation costs (Wang et al. 2020; Qian et al. 2020; Durieux et al. 2020). As a result, publicly available datasets are very scarce, and the number of vulnerabilities recorded in these datasets is limited, which leads to overfitting problems and lack of robustness with deep learning-based detection methods.

To address the above shortcomings, this paper proposes a multigranularity vulnerability detection method that aims to improve model robustness and capture more code semantics. The main features of the method are described as follows:

1. In existing studies, graph structures transformed based on source code usually retain only part of the code semantics, limiting the possible detection types. In this paper, we enrich the semantics of the code contained in the graph by converting the smart contract source code into a heterogeneous graph fusing abstract syntax trees (ASTs), CFGs, and DFGs. This approach extracts features of smart contract source code from multiple perspectives to improve the generalization ability of the model and increase the number of detectable vulnerability types.
2. The multigranularity vulnerability detection method based on heterogeneous graphs (MVD-HG) approach proposed in this paper combines the attention mechanism with a graph convolution operation for heterogeneous graphs to realize multigranularity vulnerability detection at the graph and node levels. These two levels correspond to the contract-level and line-level vulnerability detection tasks in smart contract vulnerability detection. The detection effect of the proposed method is significantly better than that of the current state-of-the-art methods, providing a more comprehensive and effective solution for smart contract vulnerability detection. Thus, complex rules do not need to be set for each attack scenario, and

fully automated detection can be achieved for different attack patterns.

3. In previous experimental work, the size of the datasets used was usually small, and the experimental results were susceptible to problems such as overfitting, reducing the reliability of the experimental results. To address this issue, a data augmentation method that expands the size of the existing dataset based on code generation methods used in related fields is implemented to improve the reliability of the experimental results and the robustness of the model.

The rest of the paper is organized as follows: "Research background" section reviews the research related to this work. In "Background" section presents a partial definition of heterogeneous graphs and justifies this definition with an example. In "Methodology" section describes the detailed design of the MVD-HG approach. In "Experiment" section discusses the experimental setup and the results, showing the effectiveness of the proposed detection method. Finally, Section 6 concludes the paper and discusses its limitations and future prospects.

Research background

Smart contract vulnerability detection

Due to the extremely challenging nature of smart contract vulnerability detection, it is impractical to rely on only manual detection methods. Thus, researchers have proposed three traditional methods for smart contract vulnerability detection: static analysis methods, dynamic analysis methods, and formal verification methods (Praitheeshan et al. 2019). In addition to traditional analysis methods, deep learning-based approaches are becoming increasingly popular in the field of smart contract vulnerability detection. However, deep learning-based methods usually need to be implemented with large-scale high-quality datasets (Rameder 2021).

Static analysis methods involve analysing program code in non-runtime environments, allowing the source code or bytecode of the target contract to be examined without executing the program. Static analysis approaches are widely used in the field of smart contract vulnerability detection. For example, Oyente (Luu et al. 2016) is based on a symbolic execution approach; Securify (Tsankov et al. 2018) is a lightweight and extensible method; SmartCheck (Tikhomirov et al. 2018) converts Solidity source code into intermediate representations based on xml format and verifies the results with XPath; Ether-Trust (Grishchenko et al. 2018) is an automated static analysis tool for bytecode-level verification; and Smart-ConDetect (Jeon et al. 2021) extracts code fragments from Solidity smart contracts for further inspection using a pretrained Bert model.

Dynamic analysis methods include fuzzy testing methods; the main idea of fuzzy testing is to observe whether the target program behaves abnormally for randomly generated inputs using many random inputs and to identify inputs that cause issues through random collisions (Hu et al. 2021). For example, Liu et al. (2018) proposed ReGuard, a fuzzy analyser for analysing reentrant vulnerabilities that performs fuzzy testing based on smart contracts by iteratively generating random but different transactions. In addition, ContractFuzzer, proposed by Jiang et al. (2018), generates random inputs based on the ABI of smart contracts and records the execution results based on these inputs; then, security analysis is performed using predefined rules.

Formal verification methods are used to validate code logic and ensure that the code does not include unanticipated errors in arbitrary situations. In contrast, traditional test-based approaches always suffer from the impossibility of testing every input and the possibility of unanticipated inputs causing anomalies in the system (Murray and Anisi 2019). Formal verification methods determine the security of the target contract by verifying that the program contains vulnerabilities that can be exploited through rigorous logical proofs. There are fewer formal verification methods than static and dynamic analysis methods. Some examples include ZEUS (Kalra et al. 2018), an automated verification framework using abstract interpretation and symbolic model checking, and KEVM (Hildenbrandt et al. 2018), a formal specification based on EVM, a reference interpreter, which is a tool for program analysis and verification based on the K framework.

Several studies have investigated deep learning-based approaches for vulnerability detection in smart contracts. For example, Peculiar (Wu et al. 2021) performed DFG and CDFG extraction with an AST transformed from source code, followed by detection with a pretrained model. In addition, Zhuang et al. (2021) transformed the source code into a custom contract graph, highlighted important nodes via normalization to obtain important information, and performed a smart contract vulnerability detection task. Liu et al. (2021) combined custom expert rules with a graph detection method to further improve the detection results. Qian et al. (2020) proposed to utilize bidirectional long-short term memory networks combined with an attention mechanism for vulnerability detection. Zhao et al. (2021) proposed a reentrant vulnerability detection method based on code embedding and the GAN model. Nguyen et al. (2022) proposed MANDO to generate heterogeneous graphs based on smart contract source code and captured code semantics in source code through metapaths for multigranularity code vulnerability detection.

Source code representation

To use deep learning methods to detect code vulnerabilities, the source code needs to be represented as a vector. Methods combining IR and machine learning have been considered effective to transform source code into vectors. For example, BGNN4VD (Cao et al. 2021) extracts syntactic and semantic information from source code with ASTs, control flow graphs, and data flow graphs. Furthermore, SedSVD (Dong et al. 2023) learns semantic and syntactic information from source code using code property graphs (CPGs) and selects several central nodes in the CPGs to construct target subgraphs. Zhang et al. (2022) constructed super dataflow graphs covering all real dataflow paths and detected whether a buffer overflow attack occurred by performing recursive dataflow analysis based on the program's super dataflow graph. Wang et al. (2020) constructed FA-AST, increased the amount of control flows and data flows in the AST to obtain more detailed code fragment information, and then performed the vulnerability detection task of code cloning.

Graph neural networks

In recent years, with the rapid development of graph neural network techniques such as GCN (Kipf and Welling 2016), GAT (Veličković et al. 2017), and HAN (Wang et al. 2019), models for understanding complex graphs have become easier to achieve. According to the complexity of heterogeneous graphs and the richness of information, HGAT (Yang et al. 2021) learns the importance of different nodes and edges in the graph through an attention mechanism to effectively capture the rich structural and semantic information in the graph. In RGCN (Schlichtkrull et al. 2018), relational features are introduced through node representations to improve the expressiveness of the model, and attention mechanisms and convolution operations are used to aggregate information. Typically, these models have better vulnerability detection capability than detection tools based on traditional methods. The difference, however, is that these methods identify the fitted relationships based on the dataset through extensive training, and sufficient data are required to obtain better results based on the validation set.

Background

In this paper, we describe how to convert smart contract source code into heterogeneous graphs based on ASTs, CFGs, and DFGs and explore data augmentation based on heterogeneous graphs as well as multigranularity vulnerability detection tasks. First, we introduce the basics of the fallback mechanism in smart contracts, which is important to understand some practices in the conversion process of heterogeneous graphs. During the conversion process, a fallback node needs to be actively added to simulate the fallback function of the attack contract, which cannot be automatically generated by building the AST.

Fallback

The fallback function is a special function in the Solidity language (ethereum 2022; Fan et al. 2021; Li 2023). This function plays an important role in smart contracts, as the function takes no arguments and returns either no value or only a Boolean value representing the success of the transaction execution. When a contract receives an undefined function call, the fallback function ensures that the contract continues to function properly, which is essential to ensure the proper execution of Solidity programs. The fallback function can also be used to receive transfer transactions with ether, and when the executing program calls the transfer function, the target contract automatically calls the fallback function to process the transaction. For example, in the attack case presented in Section 3.3, the fallback mechanism was utilized to launch the attack. Therefore, to ensure that the heterogeneous graph is consistent with the source code semantics, the fallback mechanism is introduced in the conversion process with the corresponding data flow and control flow expansions.

Formal definitions

Definition 1 (AST) The source code $SC = \{L_1, \ldots, L_n\}$ can be formally represented as $G_{AST} = \{V_{AST}, E_{AST}, \alpha_{AST}, \beta_{AST}\}$ after preprocessing. V_{AST} is the set of all nodes in the AST, denoted as $V_{AST} = \{V_1, V_2, \ldots, V_n\}$, and the subscript denotes the ordinal number of the node. Each node represents a different syntactic structure in *SC*, such as expressions, statements, and variable declarations. Different node types correspond to various syntactic structures. E_{AST} is the set of all edges in the AST, which can be formalized as $E_{AST} = \{e_{1,2}, e_{1,3}, \ldots, e_{n,m}\}$ for a set of directed edges. Here, $e_{i,j} = (V_i, V_j)$ represents the existence of an edge between the ith and jth nodes in the AST. α_{AST} denotes the mapping function between a node and the corresponding line of code, e.g., $\alpha_{AST}^{V_m} = 3$ denotes that the mth node is located on the 3rd line of the source code. β_{AST} denotes a mapping function that maps different node types, e.g., $\beta_{AST}^{V_m} = FunctionDefinition$ represents that the mth node is of type FunctionDefinition. In addition, T_{AST} is defined in this paper to describe the set of all possible node types in the AST.

Definition 2 (CFG) The heterogeneous graphs of the generated control flow graph and the AST can be formally represented by G_{HCFG}={$V_{HCFG}, E_{HCFG}, \alpha_{HCFG}, \beta_{HCFG}$}. The build process is based entirely on V_{AST}. However, since the automatically converted AST does not contain a corresponding fallback mechanism, it is necessary to manually add nodes of the fallback type. In this paper, $V_{fallback}$ is defined as the fallback node, i.e., $V_{HCFG} = V_{AST} \bigcup \{V_{fallback}\}$, and $T_{HCFG} = T_{AST} \bigcup \{fallback\}$. $E_{HCFG} = \{e_{1,2}, e_{1,3}, \ldots, e_{n,m}\} \bigcup E_{AST}$ denotes the set of directed edges in the current graph. $e_{i,j} = (V_i, V_j)$ represents the existence of a control flow edge between the ith and jth nodes in the heterogeneous graph, and E_{AST} denotes the original directed edge in the AST. The α_{HCFG} and β_{HCFG} in G_{HCFG} are defined similarly as the corresponding terms in G_{AST}.

Definition 3 (*DFG*) The data flow expansion result is similar to the control flow generator result and can be expressed formally as $G_{HDFG} = \{V_{HDFG}, E_{HDFG}, \alpha_{HDFG}, \beta_{HDFG}, \gamma_{HDFG}\}$. Because some global variables such as msg.sender and block.timestamp do not reflect their origin when building the AST, V_{block} and V_{msg} are added to complete the data flow; thus, $V_{HDFG} = V_{HCFG} \bigcup \{V_{block}, V_{msg}\}$. $E_{HDFG} = \{e_{1,2}, e_{1,3}, \ldots, e_{n,m}\} \bigcup E_{HCFG}$ denotes the set of directed edges in the heterogeneous graph. $e_{i,j} = (V_i, V_j)$ represents the existence of a data flow edge between the ith and jth nodes in the heterogeneous graph. E_{HCFG} denotes the information of the two underlying edges contained in the heterogeneous graph composed of the control flow graph and the AST. The α_{HDFG} and β_{HDFG} in G_{HDFG} are defined similarly as the corresponding terms in G_{AST}. γ_{HDFG} denotes a mapping function that maps different edge types, e.g., $\gamma^{e_{i,j}}_{HDFG} = \{AST\}$ represents that the edge between (V_i, V_j) is created when the AST is created. $ET_{HDFG} = \{AST, CFG, DFG\}$ is defined to represent all possible edge types, and $\gamma^{e_{i,j}}_{HDFG} \subseteq ET_{HDFG}$. $T_{HDFG} = T_{HCFG}$ is defined to describe the set of all possible node types in the heterogeneous graph.

Attack example

For the acquired smart contract source code, the edge information is automatically expanded on the basis of the AST nodes to generate a heterogeneous graph with richer semantic information. In this paper, the detection task is divided into two subtasks: first, the entire contract is classified with binary results to determine whether there are vulnerabilities; then, each line in the source code is classified with binary results to identify whether there are vulnerabilities in that line of code. These two detection methods satisfy the requirements in most scenarios.

The advantages of expanding the data flow and control flow based on an AST can be illustrated by an example. The left side of Fig. 1 shows source code with a reentrancy attack, with the vulnerability located in the withdraw function between lines 10 and 15 in the source code. The right side of Fig. 1 shows the transformed source code to extract the subgraph portion of the heterogeneous graph associated with the withdrawal function. In this diagram, each node represents a syntactic structure in the program code, such as expressions, statements, and variable declarations. For example, *FunctionDefinition* represents the function definition in line 10, and *amountSource* represents the corresponding amount variable in line 10. AST, CF, and DF are the edges defined in Definition 1, Definition 2, and Definition 3, respectively. AST represents the edge in the abstract syntax tree, which is used

```
1  pragma solidity 0.4.24;
2
3  contract SimpleDAO {
4      mapping (address => uint) public balance;
5
6      function donate(address to) payable public {
7          balance[to] += msg.value;
8      }
9
10     function withdraw(uint amount) public {
11         if (balance[msg.sender]>= amount) {
12             require(msg.sender.call.value(amount)());
13             balance[msg.sender]-=amount;
14         }
15     }
16
17     function queryCredit(address to) view public returns(uint) {
18         return balance[to];
19     }
20 }
```

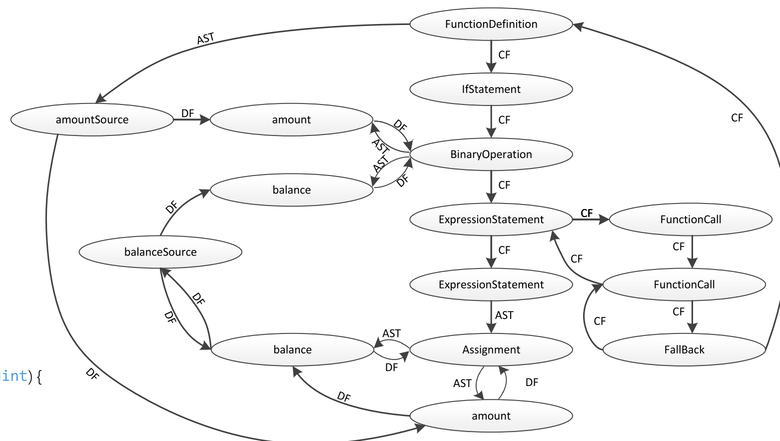

Fig. 1 Sample code transformed into a heterogeneous graph subgraph

to indicate the connection between syntax structures; CF represents the edge in the CFG, which is used to indicate how the control flow moves and executes between different nodes; and DF represents the edge in the DFG, which indicates how variables are defined or assigned between different nodes.

First, the details of the attack are assessed at the code level. The attacker contract calls the donate function to deposit startup funds for the victim contract. Then, the attacker contract calls the withdraw function to launch the attack. Since the account balance is sufficient, the IF condition in line 11 of the victim contract is judged to be true, allowing the execution of the transfer function code in line 12. Then, the fallback function in the attacker contract is called. If there is a function call to withdraw in the attacker's fallback function, the attacker can perform repeated withdrawals with recursive functions. At this point, the account balance deduction code in line 13 of the victim's contract is not executed to ensure that the result is true each time the IF conditional judgement statement in line 11 is executed, thus realizing the reentrant attack.

Next, the whole process of the withdrawal function call is evaluated from the perspective of a heterogeneous graph. The attacker contract calls the withdrawal function to launch the attack, and with the control flow, the attack contract reaches the *BinaryOperation* function for the IF condition judgement. Two data flow variables are used in the judgement process: *amountSource* and *balanceSource*. Since the attacker has already deposited the initial funds, the conditional judgement result is true, and the attacker continues to execute the next *ExpressionStatement* node along the control flow. Since this node calls the transfer function, it no longer executes in the original control flow direction and instead calls the attacker's fallback function.

A benign contract slowly moves the control flow back to the withdrawal function after processing its own fallback function logic and executes the remaining logic without any issues. However, for malicious contracts, the control flow moves to *FunctionDefinition* and starts a new round of withdrawal function calls. During this process, the account balance deduction operation is temporarily ignored, and *balanceSource* and *amountSource* are not updated through the data stream. Therefore, during each malicious call to the withdrawal function, these two variables remain unchanged until a certain fallback function returns the control flow to the normal withdrawal function.

Methodology

The general framework proposed in this paper is shown in Fig. 2 and consists of the following three parts:

1. **Heterogeneous graph construction phase** The AST is extracted based on the source code, and the heterogeneous graph of the target structure is constructed by expanding the edge relations of the nodes in the abstract syntax tree according to the logic of the source code.
2. **Data augmentation phase: this step is optional** The code fragments that do not contain vulnerabilities

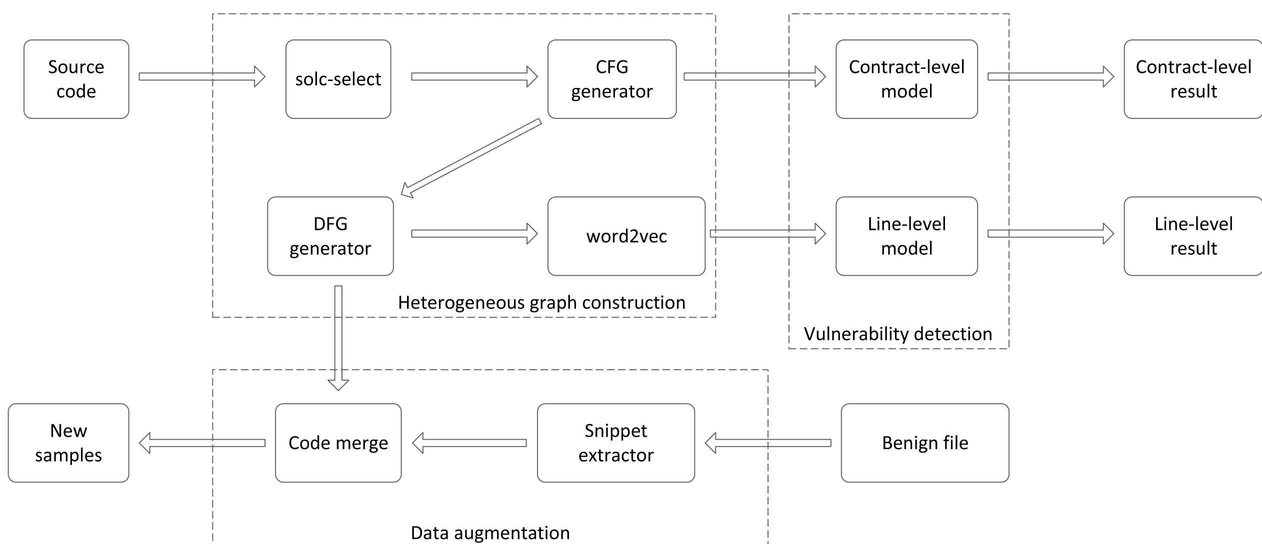

Fig. 2 General architecture diagram

are extracted from the benign contract and inserted into the contract that contains the vulnerability.

3. **Vulnerability detection phase** Based on the word-2vec pretrained word vector model, the heterogeneous graphs are vectorized into trainable graph embeddings and combined with a neural network to output the detection results.

Heterogeneous graph construction

According to previous research results on related codes, processing source code as natural language text may cause the rich semantic information in source code to be ignored (Zhang et al. 2019). In contrast, ASTs contain both the corresponding syntactic information and a certain amount of semantic information. Therefore, various methods based on ASTs have been proposed (Wu et al. 2021; Wang et al. 2020; Zhang et al. 2019; Liang et al. 2019), which help to improve the understanding of the source code by the model and thus improve the experimental results. In this paper, solc-select (Crytic 2022) is used to extract the basic AST structure based on the smart contract source code. The obtained AST is discussed in Definition 1.

Control flow generator Control flow analysis is a traditional software analysis technique that involves modelling and analysing conditional judgement statements, looping statements, and function call relationships. Control flow analysis methods can vary significantly among different programming languages. In particular, there are features in the Solidity language for writing smart contracts that are not present in other common languages. For example, the fallback function is called when the target of the function call is does not exist or the current transaction involves ether transactions.

In this paper, a connection between each node in G_{AST} and each line in the source code is established based on the content and type. The task of expanding the control flow information based on the abstract syntax tree is achieved by statically scanning the source code and simulating the program execution path. In this paper, the types of newly expanded edges are not distinguished, either in terms of normal advancement between upper and lower lines of code or in terms of call and return relationships between functions, which are reflected in the same kinds of directed edges in the heterogeneous graph. The obtained heterogeneous graph composed of the AST and CFG is discussed in Definition 2.

Data flow generator Data flow analysis is similar to control flow analysis and is another traditional software analysis technique. It is often used to analyse the flow of variable values in programs over different execution paths. Since the analysis is based on the execution path, the process is performed after the control flow generator. The approach used in this paper is forwards analysis, which starts at the entry point of the program and traverses the execution path generated by the control flow generator until the exit point of the program. The execution of a smart contract also involves many global variables, such as *msg.sender*, representing the initiator of the transaction, and *block.timestamp*, representing the timestamp of the current block. The origin of these variables is not reflected in the abstract syntax tree, and similar to the fallback function, they need to be actively added to the abstract syntax tree.

In data flow analysis, assignment and referencing of variables are both considered ways of passing data flows. Similarly, in this paper, no distinction needs to be made between the different types, which are also reflected in the same directed edges in the heterogeneous graph. In this paper, how the variables flow through the program is considered rather than the specific values represented by the variables, as this approach reflects the method by which hackers attack contracts. The obtained heterogeneous graph consisting of the AST, CFG and DFG is discussed in Definition 3.

Thus far, the data flow and control flow in the smart contract have been fused with the AST and used to enrich the graph information to be trained to improve the classification ability of the model. Different from the idea of Liu et al. (2021), the idea in this paper is that the intricate edge relations and the structural information are important for detecting various vulnerabilities, and no deletion operation should be performed to simplify the graph structure, which would lead to the loss of code semantics. This heterogeneous graph can increase the generalization ability of the model, thus improving its detection ability for various types of vulnerabilities.

Data augmentation

The lack of datasets for this task has been criticized, as the number of datasets available for training is very limited and the cost of manual labelling is high. Furthermore, the trained models show poor performance, and the validation sets are too small to fully verify the validity of the models.

In the field of computer vision, it is common to enhance the robustness or accuracy of models by data augmentation (Wong et al. 2016), and the most common data enhancement strategies include distortion, adding noise, rotation, panning, clipping, and merging. In this paper, we propose a data augmentation method for smart contract source code, aiming to address the issues caused by insufficient training data. Specifically, a function is selected from the benign file, and the remaining necessary code required for the function to run is determined

by control flow analysis. Then, the benign function is inserted before the line containing the vulnerability in the code to be enhanced. To ensure that the heterogeneous graph is changed without affecting the original code execution results, the inserted part is wrapped using a while loop (false). The remaining necessary code is inserted into a blank position in the file to be enhanced according to the dependency with the benign function, thus generating a new experimental sample. Finally, the line-level and contract-level labels of the new samples are determined according to the inserted positions. This approach can effectively expand the dataset and improve the generalization ability and robustness of the model. In terms of the graph structure, data augmentation methods connect two partial subtrees of an abstract syntax tree by controlling the flow edges. Theoretically, enough training data can be constructed with data augmentation methods to alleviate the training and validation problems.

The specific enhancement process is shown in Algorithm 1. The algorithm inputs include srcPath (path of the complete contract file to be enhanced), segPath (path of the benign code file), nDict (dictionary stored by node type after converting the benign file to the AST) and insertLine (insert location in the file to be enhanced).

Algorithm 1 Data augmentation

Input: srcPath, segPath, nDict, insertLine
Output: new sample files and their labels

1: $fullNodes \Leftarrow set()$
2: **for** $node \in nDict[FunctionDefinition]$ **do**
3: $\quad fullNodes \Leftarrow fullNodesByCDFG(node)$
4: $\quad$ **for** $baseNode \in BaseClass(node)$ **do**
5: $\quad\quad tmp \Leftarrow FullNodesByCDFG(node)$
6: $\quad\quad fullNodes.append(tmp)$
7: $\quad$ **end for**
8: $\quad segLine, srcLine \Leftarrow DetailMsg(srcPath, segPath, insertLine, nDict)$
9: $\quad diffLine \Leftarrow \{\}$
10: $\quad$ **for** $i = 0 \Rightarrow len(srcLine)$ **do**
11: $\quad\quad flag \Leftarrow (srcLine[i] == -1)$
12: $\quad\quad flag = flag\&(srcLine[i+1] == -1)$
13: $\quad\quad diffLine \Leftarrow recordDiff(segLine[i], srcLine[i], flag)$
14: $\quad\quad i \Leftarrow i + 2$
15: $\quad$ **end for**
16: $\quad updateLabel(diffLine, srcPath, segPath)$
17: **end for**

In the algorithm, the set fullNodes is first initialized to record all the nodes that may be involved in the execution of the function. In line 2, all nodes with node type FunctionDefinition are traversed, i.e., all functions in the benign file are traversed. In lines 3–7, all nodes that may be involved in executing the function are added to the fullNodes set to map the specific lines in the benign file that need to be inserted into the new sample. In line 8–9, the DetailMsg function calculates the order in which the two files should be saved as new experimental samples when they are merged, and the results are stored as arrays as segLine and srcLine. In lines 10–15, the benign file and the file to be enhanced are fused according to segLine and srcLine. Then, diffLine is updated to record the deviation between the position of the vulnerability in the file to be enhanced and its position in the new sample. In line 16, the tag of the new sample file is updated using diffLine.

Vulnerability detection

Vectorizing graph data is one of the important steps in using graph neural networks for deep learning. Another idea proposed in this paper is to use a word vector model to map the content and type of each node in the AST and thus obtain embeddings that can be used in the neural network. The word2vec (Mikolov et al. 2013, 2013) method is used as an intermediate step between the raw graph data and trainable node embeddings in this paper. Word2vec is a neural network-based natural language processing technique for converting text into vector representations. Since code can be considered a special form of text, a similar technique can be used to convert codes into vector representations. Specifically, elements such as identifiers, variables, functions, and operators in the code can be considered as words and trained using a word2vec model to obtain vector representations of each word. For example, Lin et al. (2018) used word2vec to map each element in a serialized form of an AST to a trainable vector. In this paper, we first serialize the AST structure using depth-first search (DFS) and breadth-first search (BFS) techniques to preserve the structural properties of the AST and convert each node in the serialized AST to a string containing the node type and node content. This serialized and converted corpus is used for word vector model training. Thus, word2vec can be used to extract the semantic relations between different nodes in the AST. Then, these relations can be converted into vector representations to better represent the similarities and differences between nodes, thereby allowing the model to better interpret and understand the relations between nodes. The vectorization process is discussed in Definition 4.

Definition 4 (Vectorization Process) The program source code $SC = \{L_1, L_2, \ldots, L_n\}$ represents each line of statements in the source file. L_i consists of an ordered set

of symbols, e.g., $L_i = \{S_{i,1}, S_{i,2}, ..., S_{i,m}\}$ indicates that the ith line in the source code contains m symbols. Moreover, L_i can be represented by a set of nodes in G_{HDFG}, e.g., $L_i = \{V_u, V_{u+1}, ..., V_w\}$, where each node may contain one or more symbols, so $|w-u| \leq m-1$. The process to transform V_k into a vector is shown in Eq. 1. $h^0_{V_k}$ represents the original input to node V_K, the dimension is $R^{1\times d}$, and d denotes the vector length specified after the word2vec transformation.

$$h^0_{V_k} = word2vec(V_k) \oplus word2vec(\beta^{V_k}_{HDFG}) \tag{1}$$

$$h^{l+1}_{V_i} = \sigma(\sum_{r\in ET_{HDFG}} \sum_{j\in N^r_i} \frac{1}{C_{i,r}} W^l_r h^l_j + W^l_0 h^l_i) \tag{2}$$

The classifiers used in this paper are divided into two categories: contract-level vulnerability detection and line-level vulnerability detection. Contract-level vulnerability detection is the most basic classification task in which binary results are generated to indicate whether a contract is vulnerable. Specifically, the model performs the node classification task via RGCN, which aggregates the information of neighbouring nodes through a multilayer convolution operation (Schlichtkrull et al. 2018). The aggregation process is shown in Eq. 2, where N^r_i denotes the set of neighbour nodes of node i that have relationship r with node i, and σ is the ReLU function.

After several rounds of RGCN aggregation, it is necessary to divide the nodes according to the corresponding contract and perform an OR operation based on the division results to obtain the final detection results. Equation 3 defines the process of computing the contract-level results, where $\widehat{y_{cs}} \in \{0,1\}$ indicates the presence or absence of vulnerabilities in the source file. T_{cs} denotes the set of all contracts in the file, and V^{con}_C represents the set of all nodes belonging to contract type *con*:

$$\widehat{y_{cs}} = \bigcup_{con\in T_{cs}} sigmoid(W_{cs} \frac{1}{|V^{con}_C|} \sum_{j\in V^{con}_C} h_{V_j} + b_{cs}) \tag{3}$$

Line-level vulnerability detection is a more granular process than contract-level vulnerability detection, and the outputs are the binary detection results for each line in the source code. In this task, RGCN is also used to aggregate information from surrounding neighbouring nodes, and all nodes associated with each line are pooled equally to obtain the embedding of the source code for that line. Based on these line-level embeddings, the model uses an attention mechanism to further fuse the contextual information. The embeddings are mapped to binary detection results for each line using a fully connected layer and an activation function. The process of RGCN aggregation is shown in Eq. 2. Equation 4 defines the process of computing the line-level results. V^l_L denotes the set of all nodes located in line l, and *cof*(k) denotes the attention factor assigned to the kth line of source code.

$$\widehat{y_{L_k}} = sigmoid(W_{line} \sum_{l=k-1}^{k+1} \sum_{j\in V^l_L} h_{V_j} \frac{cof(l)}{\sum_{i=k-1}^{k+1} cof(i)} + b_{line}) \tag{4}$$

Experiment

This section presents the experimental setup and experimental results to answer the following research questions. RQ1: How does the contract-level vulnerability detection performance of the MVD-HG model compare to that of several state-of-the-art vulnerability detection tools? RQ2: How does the line-level vulnerability detection performance of the MVD-HG model compare to that of the state-of-the-art MANDO model? RQ3: Does the proposed data enhancement technique for the deep learning-based approach enhance performance? RQ4: What are the impacts of different modules in the heterogeneous graph model, including the AST, CFG, and DFG, on the experimental results?

Datasets

Four datasets were used in the experiments. The Contract-Origin dataset was used for contract-level vulnerability detection without data augmentation. The Line-Origin dataset was used for line-level vulnerability detection without data augmentation. These two datasets were derived from the experimental dataset of MANDO. Contract-augmentation and line-augmentation were the contract-level vulnerability dataset and line-level vulnerability dataset after enhancement using the data enhancement methods proposed in this paper. Each dataset contains seven vulnerability types, and Table 1 shows the details of these four datasets.

Experimental setup

In the experiments, to ensure that the original small-scale dataset is sufficient for model training and testing, the dataset was partitioned into a training set containing of 70% of the data and a testing set containing the remaining 30%. This partitioning ratio was also used for the enhanced datasets to ensure consistency. The number of epochs followed the original experiment setting and was set to 50, and each experiment was independently performed 20 times to achieve stable average results. In addition to the deep learning-based comparison method, experiments were conducted with a variety of traditional

Table 1 Detailed statistical information about the dataset

Vulnerability type	Contract-level		Line-level	
	Number of contract-origin files	Number of contract-augmentation files	Number of line-origin files	Number of line-augmentation files
Access control	114	400	57	200
Arithmetic	120	502	60	251
Denial of service	92	286	46	143
Front running	88	342	44	171
Reentrancy	142	362	71	181
Time manipulation	100	212	50	106
Unchecked low level call	190	728	95	364

tools through Smartbug (Ferreira et al. 2020), including HoneyBadger (Torres and ichen 2019), Mythril (Consensys 2022), Osiris (Torres et al. 2018), Oyente (Luu et al. 2016), Securify (Tsankov et al. 2018), Slither (Feist et al. 2019) and SmartCheck (Tikhomirov et al. 2018). HoneyBadger uses symbolic execution and heuristics to identify vulnerabilities. Mythril uses taint analysis and control flow analysis techniques to detect a variety of vulnerabilities, such as reentrancy vulnerabilities and integer overflow vulnerabilities. Osiris combines symbolic execution and taint analysis techniques and is mainly used to detect integer overflow vulnerabilities. Oyente uses a symbolic execution approach to simulate the EVM and traverse the different execution paths of the contract. Securify investigates contracts for security vulnerabilities by analysing their dependency graphs and extracting precise semantic information from the code and has the advantages of being lightweight and scalable. SmartCheck converts Solidity source code into an intermediate representation with an XML-based format and verifies the results with XPath.

We conducted the experiments on a server with an Intel Xeon E5-2680 v4 CPU (with 14 × 2.4 GHz cores), 128 GB of memory, and 4 × GTX 2080Ti GPUs running on Ubuntu 20.04. The source code and part of the datasets are available at https://github.com/Astronaut-diode/MVD-HG.

Experimental results

RQ1: How does the contract-level vulnerability detection performance of the MVD-HG model compare to that of several state-of-the-art vulnerability detection tools?

Contract-level vulnerability detection is the most common task in the field of smart contract vulnerability detection. This paper aims to use multiple smart contract vulnerability detection tools to demonstrate the potential of MVD-HG for this basic task. In this experiment, the vulnerability detection methods are divided into two categories, detection tools based on deep learning methods and traditional tools, and a timeout time of 600 s is set for the traditional tools. The experimental results of the various methods are shown in Table 2, and the F1 score, which was determined based on the Contract-Origin dataset, is used as the evaluation metric. This experiment follows the method proposed in the MANDO paper, in which four experiments were performed using four node feature generators, nodetype, meta2path, line and node2vec. The aim of these generators is the same as that of the word2vec method used in this paper, aiming to obtain trainable node features.

The following two conclusions can be drawn from Table 2. First, compared to traditional tools, deep learning-based vulnerability detection methods typically enable more comprehensive vulnerability detection and achieve better overall detection results. This indicates that traditional tools have smaller detection scopes, are less likely to detect unanticipated vulnerability types, and need to continuously produce new heuristic rules for new attacks to achieve vulnerability detection. In contrast, the deep learning-based approach is more adaptable and can better handle new types of vulnerabilities. Second, the MVD-HG approach outperformed MANDO based on the Contract-Origin dataset. This indicates that the heterogeneous graphs used in the MVD-HG model capture richer semantic information, enhance the model's understanding of the program code, and improve the effectiveness of contract-level vulnerability detection tasks.

RQ2: How does the line-level vulnerability detection performance of the MVD-HG model compare to that of the state-of-the-art MANDO model?

The experiment to address RQ2 uses the same configuration as that to address RQ1 and use the Line-Origin dataset to explore the effectiveness of MVD-HG for the line-level vulnerability detection task. The results of this experiment are shown in Table 3. For each line in Table 3, to obtain the results of the traditional tool, the researchers determined the predicted results for each line of

Table 2 Original contract-level vulnerability detection results

Testing method		Access control	Arithmetic	Denial of service	Front running	Reentrancy	Time manipulation	Unchecked low level call
Traditional tools	HoneyBadger	–	–	–	–	0.4222	–	–
	Mythril	0.4249	0.5069	–	–	0.3695	–	0.5648
	Osiris	–	0.4303	0.0425	–	0.5799	0.0769	–
	Oyente	–	0.4042	0.0425	–	0.5656	–	–
	Securify	0.0344	–	–	0.0416	0.4421	–	0.5757
	Slither	*0.8761*	–	0.1960	–	*0.9523*	**0.9504**	0.6363
	SmartCheck	0.6744	*0.8679*	0.0833	–	**0.9857**	*0.9473*	0.6896
MANDO	nodetype	0.7119	0.6685	0.8737	0.8731	0.7609	0.8503	*0.7208*
	meta2path	0.5770	0.5284	0.6016	0.6219	0.5506	0.5947	0.5137
	line	0.6512	0.5491	*0.8915*	*0.8986*	0.7104	0.8771	0.5944
	node2vec	0.5571	0.6411	0.8386	0.8605	0.7139	0.7338	0.6610
MVD-HG		**0.9355**	**0.9196**	**0.9146**	**0.9559**	0.9397	0.9351	**0.9056**

The best two average results are shown in bold and italic fonts

Table 3 Original line-level vulnerability detection results

Testing method		Access control	Arithmetic	Denial of service	Front running	Reentrancy	Time manipulation	Unchecked low level call
Traditional tools	HoneyBadger	–	–	–	–	–	–	–
	Mythril	0.1137	0.0247	–	–	0.0135	–	0.0324
	Osiris	–	0.0247	–	–	0.0062	0.0003	–
	Oyente	–	0.0061	–	–	0.0062	–	–
	Securify	0.0004	–	–	–	–	–	–
	Slither	**0.9197**	–	0.0007	–	0.3872	0.5972	0.0355
	SmartCheck	0.0379	0.0614	–	–	0.0691	0.2112	0.0442
MANDO	nodetype	0.7721	0.8162	0.7983	0.8819	0.8424	0.8664	0.6595
	meta2path	0.6797	0.7484	0.6722	0.8608	0.7603	0.7381	0.5071
	line	0.8119	0.8158	*0.8212*	0.9047	0.8627	0.8921	0.8337
	node2vec	0.8198	*0.8435*	0.8209	*0.9051*	*0.8640*	*0.9029*	*0.8481*
MVD-HG		*0.8269*	**0.9312**	**0.8302**	**0.9105**	**0.9469**	**0.9225**	**0.9015**

The best two average results are shown in bold and italic fonts

source code in the target file and compared the results with the original tags.

The experimental results show that most traditional tools perform poorly for line-level vulnerability detection tasks, and a deep understanding of the program code is lacking, making it difficult to obtain fine-grained vulnerability detection results. In contrast, the AST-based detection methods, such as MANDO and MVD-HG, can provide a more comprehensive understanding of the source code and achieve better detection results. In addition, MVD-HG outperforms MANDO in line-level vulnerability detection tasks, indicating that the use of heterogeneous graphs and attention mechanisms can improve the model's understanding of the program code and the vulnerability detection results.

RQ3: Does the proposed data enhancement technique for the deep learning-based approach enhance performance?

To address RQ3, researchers conducted experiments to validate the effectiveness of the proposed data enhancement method for both contract-level and line-level vulnerability detection data. Specifically, experiments were conducted based on the Contract-Origin and Contract-Augmentation datasets for the contract-level task and the Line-Origin and Line-Augmentation datasets for the

line-level task. The experimental results are shown in Tables 4 and 5.

In Table 4, each pair of rows represents the results for one kind of vulnerability data before and after data enhancement, and two main conclusions can be drawn from the experimental results. First, the performance of the traditional tool based on the augmented vulnerability datasets does not show a simple monotonic increasing or decreasing trend. This result is consistent with the fact that the traditional tool is based on expert rules, and increasing the amount of data in the dataset does not necessarily lead to improved classification results. This indicates that the data augmentation method proposed in this paper does not arbitrarily add simple new samples and that the added samples are difficult to identify. Second, for the deep learning-based method, the performance was improved to some degree in all experiments, except for the MANDO experiment using meta2path as the node feature generator, which showed a reduced prediction effect. These results indicate that there is a bottleneck in capturing the semantic relationships between different nodes in the AST using meta2path and that relying on only metapath will lead to the loss of semantic information in the source code. These results also demonstrate that the data augmentation method for smart contract samples proposed in this paper is effective and

Table 4 Enhanced contract-level vulnerability detection results

Testing method		Access control	Arithmetic	Denial of service	Front running	Reentrancy	Time manipulation	Unchecked low level call
Traditional Tools	HoneyBadger	–	–	–	–	0.1809	–	–
	Mythril	0.4855	0.5365	–	–	0.5426	–	0.4739
	Osiris	–	0.4714	0.1428	–	0.5041	0.2877	–
	Oyente	–	0.4016	0.1428	–	0.4916	–	–
	Securify	0.0099	–	–	0.1176	0.4463	–	0.6212
	Slither	*0.8647*	–	0.1059	–	0.7928	*0.9056*	0.5333
	SmartCheck	0.5964	0.8298	0.3372	–	**0.9802**	0.8287	0.7033
MANDO	nodetype	0.7562	*0.8431*	*0.8422*	*0.9103*	0.8267	0.8018	*0.8461*
	meta2path	0.5984	0.5699	0.5324	0.5660	0.5695	0.5641	0.5717
	line	0.7095	0.8108	0.7946	0.8311	0.7419	0.7647	0.7947
	node2vec	0.7343	0.8198	0.7785	0.8184	0.6723	0.7103	0.8300
MVD-HG		**0.9430**	**0.9461**	**0.9670**	**0.9551**	*0.9564*	**0.9518**	**0.9775**

The best two average results are shown in bold and italic fonts

Table 5 Enhanced line-level vulnerability detection results

Testing method		Access control	Arithmetic	Denial of service	Front running	Reentrancy	Time manipulation	Unchecked low level call
Traditional tools	HoneyBadger	–	–	–	–	–	–	–
	Mythril	0.1592	0.0501	–	–	0.0167	–	0.0274
	Osiris	–	0.0289	–	–	0.0044	0.0018	–
	Oyente	–	0.0072	–	–	0.0044	–	–
	Securify	0.0001	–	–	0.0009	–	–	–
	Slither	**0.9633**	–	0.0020	–	0.3930	0.6081	0.0310
	SmartCheck	0.0293	0.0647	–	–	0.0677	0.2122	0.0459
MANDO	nodetype	0.7407	0.7838	0.7832	0.8541	0.8195	0.8207	0.6678
	meta2path	0.6078	0.7164	0.6100	0.8567	0.7405	0.6784	0.5563
	line	0.7542	0.8165	*0.8064*	*0.9013*	0.8457	0.8759	0.8398
	node2vec	0.8048	*0.8306*	0.8042	0.9006	*0.8540*	*0.8799*	*0.8508*
MVD-HG		*0.9544*	**0.9806**	**0.9493**	**0.9653**	**0.9747**	**0.9551**	**0.9748**

The best two average results are shown in bold and italic fonts

can produce more training data, allowing the model to improve its detection capability.

As seen in Table 5, there are fluctuations in the detection effectiveness of the traditional tools before and after data augmentation in the line-level vulnerability detection task. This indicates that the data augmentation method is equally effective in the line-level vulnerability detection task, in which the additional data samples are all difficult to detect. Notably, in the MANDO experiments, the data enhancement method not only did not improve the detection effect but also led to a slight decrease or no substantial change. However, MVD-HG showed significantly improved performance after data enhancement. This indicates that there is some overfitting problem with the MANDO approach in the line-level vulnerability detection experiments. The MVD-HG model, on the other hand, shows more robust performance in the line-level vulnerability detection task, which is attributed to the three-graph fusion to create the heterogeneous graphs and the use of an attention mechanism to reference the contextual information.

RQ4: What are the impacts of different modules in the heterogeneous graph model, including the AST, CFG, and DFG, on the experimental results?

To address RQ4, several ablation experiments are conducted in this paper to demonstrate the effectiveness of transforming the smart contract source code into a heterogeneous graph composed of ASTs, CFGs, and DFGs in the vulnerability detection task and to explore the roles of the AST, CFG, and DFG in the model. The Contract-Augmentation and Line-Augmentation datasets were used for the experiments, and the results are shown in Table 6.

Based on the experimental results in Table 6, two conclusions can be drawn. First, the CFG is a higher priority for two attacks, reentrancy and time manipulation, and is sufficient to capture the code semantics needed to identify these two attacks; however, this component shows significant performance degradation for the other five attacks. This suggests that different heterogeneous graph components have distinct effects on various vulnerability types that may be related to the characteristics of the vulnerability. Second, changing the components in the heterogeneous graph can slightly improve the performance in certain vulnerability detection tasks. For example, merging the AST and DFG into a heterogeneous graph slightly improves performance in the contract-level arithmetic, denial of service, front running, and time manipulation vulnerability detection tasks but tends to lead to significant performance degradation for the other vulnerability types. Thus, although merging the AST, CFG, and DFG into heterogeneous graphs may introduce some additional information, the approach successfully improves the overall performance. Additionally, the row of Average Runtime indicates the time required for training a single file (in seconds). It can be seen that different combinations of heterogeneous graphs have little effect on training time. Given the combination of AST,

Table 6 Results of ablation experiments

Task	Vulnerability type	AST CFG DFG	CFG DFG	AST DFG	AST CFG	AST	CFG	DFG
Contract level	Access control	0.9430	0.7797	0.8800	0.9438	0.8564	0.6867	0.6697
	Arithmetic	0.9461	0.9563	0.9558	0.9667	0.9537	0.8968	0.8978
	Denial of service	0.9670	0.9260	0.9781	0.9763	0.9538	0.7668	0.7845
	Front running	0.9551	0.9790	0.9658	0.9627	0.9793	0.9566	0.9574
	Reentrancy	0.9564	0.9740	0.8669	0.9146	0.8453	0.9449	0.9536
	Time manipulation	0.9518	0.9548	0.9634	0.9635	0.9440	0.9064	0.9308
	Unchecked low level call	0.9775	0.7683	0.9332	0.9630	0.9513	0.7040	0.6888
	Average	**0.9567**	0.9054	0.9347	*0.9558*	0.9262	0.8374	0.8403
	Average runtime	13.34	12.75	13.60	13.51	13.80	13.19	13.01
Line level	Access control	0.9544	0.9681	0.8773	0.9457	0.8869	0.9225	0.9161
	Arithmetic	0.9806	0.9841	0.9473	0.9705	0.9435	0.9666	0.9600
	Denial of service	0.9493	0.9560	0.8732	0.9267	0.8702	0.8788	0.8825
	Front running	0.9653	0.9508	0.9273	0.9407	0.9284	0.9441	0.9445
	Reentrancy	0.9747	0.9127	0.8643	0.9135	0.8675	0.8826	0.8694
	Time manipulation	0.9551	0.9097	0.8805	0.8934	0.8813	0.9031	0.8973
	Unchecked low level call	0.9748	0.9585	0.9159	0.9713	0.9564	0.9087	0.9166
	Average	**0.9648**	*0.9485*	0.8979	0.9374	0.9191	0.9152	0.9123
	Average Runtime	87.83	90.01	91.11	87.63	84.83	88.76	89.57

The best two average results are shown in bold and italic fonts

CFG and DFG, for a source file of 40KB, contract-level vulnerability detection takes 30 s, while line-level vulnerability detection takes 72 s. Meanwhile, for a source file of 1KB, contract-level vulnerability detection can be completed within 1 s, and line-level vulnerability detection only takes 24 s. This confirms that the proposed approach to merge these three parts into a heterogeneous graph is appropriate to better understand the source code and achieve highly accurate detection results.

Conclusion

This paper proposes a method to generate hybrid heterogeneous graphs containing ASTs, CFGs and DFGs based on Solidity smart contract source code to capture more complete semantic information. The proposed method deepens the model's understanding of the source code and enables vulnerability detection at both the contract level and line level. The results of several experiments with both original and enhanced datasets show that the proposed approach is promising and outperforms the state-of-the-art methods; thus, the proposed approach is worthy of reference.

However, the method proposed in this paper can still be improved. For example, to protect the security of smart contracts, most of the contract source code is not published, and only the converted bytecode of the contract source code is released. Bytecode is not currently applicable with the methods proposed in this paper, so it is imperative to explore how to use deep learning methods for vulnerability identification based on bytecode files. In addition, most of the existing detection tools only identify which part of the source code is vulnerable but cannot determine specific attack routes; thus, the results are less interpretable. Therefore, determining the specific line of attack should be the focus of future work.

Acknowledgements
Here, we sincerely want to express our gratitude to all those who have provided help and support during the research process of this article.

Author contributions
All authors have contributed to this manuscript and approve of this submission.

Funding
This work was supported by the Major Program of Natural Science Foundation of Zhejiang Province (No. LD22F020002), the National Natural Science Foundation of China (Nos. 62372410, U22B2028), the Zhejiang Provincial Natural Science Foundation of China (No. LZ23F020011), the Fundamental Research Funds for the Provincial Universities of Zhejiang (No. RF-A2023009) and the Key R&D Projects in Zhejiang Province (No. 2021C01117).

Availability of data and materials
Our experimental data and code can be found at https://github.com/Astronaut-diode/MVD-HG.

Declarations

Competing interests
The authors declare that they have no conflict of interest.

Received: 31 October 2023 Accepted: 10 April 2024
Published online: 11 October 2024

References

Cao S, Sun X, Bo L, Wei Y, Li B (2021) BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection. Inf Softw Technol 136:106576
Consensys (2022) Consensys/mythril: security analysis tool for EVM bytecode. Supports smart contracts built for Ethereum, Hedera, Quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains. https://github.com/Consensys/mythril
Crytic (2022) crytic/solc-select: Manage and switch between Solidity compiler versions. https://github.com/crytic/solc-select
Dong Y, Tang Y, Cheng X, Yang Y, Wang S (2023) SEDSVD: statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. Inf Softw Technol 158:107168
Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 530–541
ethereum (2022) Contracts—Solidity 0.8.22 documentation. https://docs.soliditylang.org/en/latest/contracts.html#fallback-function
Fan Y, Shang S, Ding X (2021) Smart contract vulnerability detection based on dual attention graph convolutional network. In: Collaborative computing: networking, applications and worksharing: 17th EAI international conference, CollaborateCom 2021, Virtual Event, October 16–18, 2021, Proceedings, Part II 17. Springer, pp 335–351
Feist J, Grieco G, Groce A (2019) Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd international workshop on emerging trends in software engineering for blockchain (WETSEB). IEEE, pp 8–15
Ferreira JF, Cruz P, Durieux T, Abreu R (2020) SmartBugs: a framework to analyze solidity smart contracts. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, pp 1349–1352
Grishchenko I, Maffei M, Schneidewind C (2018) EtherTrust: sound static analysis of Ethereum bytecode. Technische Universität Wien, technical report, pp 1–41
Hildenbrandt E, Saxena M, Rodrigues N, Zhu X, Daian P, Guth D, Moore B, Park D, Zhang Y, Stefanescu A (2018) KEVM: a complete formal semantics of the Ethereum virtual machine. In: 2018 IEEE 31st computer security foundations symposium (CSF). IEEE, pp 204–217
Hu B, Zhang Z, Liu J, Liu Y, Yin J, Lu R, Lin X (2021) A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. Patterns 2(2)
Jeon S, Lee G, Kim H, Woo SS (2021) SmartConDetect: highly accurate smart contract code vulnerability detection mechanism using BERT. In: KDD workshop on programming language processing
Jiang F, Cao Y, Xiao J, Yi H, Lei G, Liu M, Deng S, Wang H (2022) VDDL: a deep learning-based vulnerability detection model for smart contracts. In: International conference on machine learning for cyber security. Springer, pp 72–86
Jiang B, Liu Y, Chan WK (2018) ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 259–269
Kalra S, Goel S, Dhawan M, Sharma S (2018) Zeus: analyzing safety of smart contracts. In: Ndss, pp 1–12
Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907
Li J (2023) Metamorphic testing for smart contract vulnerabilities detection. arXiv preprint arXiv:2303.03179
Liang H, Sun L, Wang M, Yang Y (2019) Deep learning with customized abstract syntax tree for bug localization. IEEE Access 7:116309–116320

Lin G, Zhang J, Luo W, Pan L, Xiang Y, De Vel O, Montague P (2018) Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans Ind Inf 14(7):3289–3297
Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B (2018) ReGuard: finding reentrancy bugs in smart contracts. In: Proceedings of the 40th international conference on software engineering: companion proceedings, pp 65–68
Liu Z, Qian P, Wang X, Zhuang Y, Qiu L, Wang X (2021) Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans Knowl Data Eng
Luu L, Chu D-H, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 254–269
Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781
Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013) Distributed representations of words and phrases and their compositionality. Adv Neural Inf Process Syst 26
Murray Y, Anisi DA (2019) Survey of formal verification methods for smart contracts on blockchain. In: 2019 10th IFIP international conference on new technologies, mobility and security (NTMS). IEEE, pp 1–6
Nguyen HH, Nguyen N-M, Xie C, Ahmadi Z, Kudendo D, Doan T-N, Jiang L (2022) MANDO: multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities. In: 2022 IEEE 9th international conference on data science and advanced analytics (DSAA). IEEE, pp 1–10
Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A (2018) Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference, pp 653–663
Praitheeshan P, Pan L, Yu J, Liu J, Doss R (2019) Security analysis methods on Ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605
Qian P, Liu Z, He Q, Zimmermann R, Wang X (2020) Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access 8:19685–19695
Rameder H (2021) Systematic review of Ethereum smart contract security vulnerabilities, analysis methods and tools
Samreen NF, Alalfi MH (2021) A survey of security vulnerabilities in Ethereum smart contracts. arXiv preprint arXiv:2105.06974
Schlichtkrull M, Kipf TN, Bloem P, Van Den Berg R, Titov I, Welling M (2018) Modeling relational data with graph convolutional networks. In: The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15. Springer, pp 593–607
Tang X, Zhou K, Cheng J, Li H, Yuan Y (2021) The vulnerabilities in smart contracts: a survey. In: Advances in artificial intelligence and security: 7th international conference, ICAIS 2021, Dublin, Ireland, July 19–23, 2021, Proceedings, Part III 7. Springer, pp 177–190
Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y (2018) SmartCheck: static analysis of Ethereum smart contracts. In: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, pp 9–16
Torres CF, ichen M (2019) The art of the scam: demystifying honeypots in Ethereum smart contracts. In: 28th USENIX security symposium (USENIX security 19), pp 1591–1607
Torres CF, Schütte J, State R (2018) Osiris: hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th annual computer security applications conference, pp 664–676
Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Buenzli F, Vechev M (2018) Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 67–82
Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y (2017) Graph attention networks. arXiv preprint arXiv:1710.10903
Wang W, Song J, Xu G, Li Y, Wang H, Su C (2020) ContractWard: automated vulnerability detection models for Ethereum smart contracts. IEEE Trans Netw Sci Eng 8(2):1133–1144
Wang X, Ji H, Shi C, Wang B, Ye Y, Cui P, Yu PS (2019) Heterogeneous graph attention network. In: The world wide web conference, pp 2022–2032
Wang W, Li G, Ma B, Xia X, Jin Z (2020) Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 261–271
Wong SC, Gatt A, Stamatescu V, McDonnell MD (2016) Understanding data augmentation for classification: when to warp? In: 2016 international conference on digital image computing: techniques and applications (DICTA). IEEE, pp 1–6
Wu H, Zhang Z, Wang S, Lei Y, Lin B, Qin Y, Zhang H, Mao X (2021) Peculiar: smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd international symposium on software reliability engineering (ISSRE). IEEE, pp 378–389
Yang T, Hu L, Shi C, Ji H, Li X, Nie L (2021) HGAT: heterogeneous graph attention networks for semi-supervised short text classification. ACM Trans Inf Syst TOIS 39(3):1–29
Zhang Y, Chen L, Nie X, Shi G (2022) An effective buffer overflow detection with super data-flow graphs. In: 2022 IEEE international conference on parallel & distributed processing with applications, big data & cloud computing, sustainable computing & communications, social computing & networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE, pp 684–691
Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 783–794
Zhao H, Su P, Wei Y, Gai K, Qiu M (2021) Gan-enabled code embedding for reentrant vulnerabilities detection. In: Knowledge science, engineering and management: 14th international conference, KSEM 2021, Tokyo, Japan, August 14–16, 2021, Proceedings, Part III 14. Springer, pp 585–597
Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q (2021) Smart contract vulnerability detection using graph neural networks. In: Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence, pp 3283–3290

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.