MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities

Anonymous Author(s)

ABSTRACT

Learning heterogeneous graphs consisting of nodes and edges of different types has recently attracted considerable attention because of their enhanced results compared to homogeneous graph techniques. However, existing techniques still fall short in handling complex heterogeneous graphs where the number of different types of nodes and edges can be large and changing, such as control-flow graphs representing possible software code execution flows. Therefore, developing techniques and tools for such graphs can be highly beneficial for detecting vulnerabilities in software and ensuring their reliability. This paper proposes MANDO, a new heterogeneous graph representation for the Ethereum smart contracts based on both control-flow graphs and call-graphs containing different types of nodes and links, called *heterogeneous contract graphs*. To capture the heterogeneous graphs' structural and semantic information, MANDO extracts customized methapaths, which compose relational connections between different types of nodes and their neighbors. Moreover, it develops a multi-metapath heterogeneous graph attention network to learn multi-level embeddings of different types of nodes and their metapaths in the heterogeneous contract graphs, which can capture the code semantics of smart contracts more accurately and facilitate the fine-grained line-level and function-level vulnerability detection. Our extensive evaluation on large smart contract datasets shows that MANDO, improves vulnerability detection results of other graph learning-based approaches at the coarse-grained contract and function levels. More importantly, it is the first learning-based approach that is capable of identifying vulnerabilities at the fine-grained *line-level*, and significantly improves the traditional code analysis-based vulnerability detection approaches by 11.35% to 70.81% in terms of F1-score.

KEYWORDS

heterogeneous graphs, graph embedding, graph neural networks, vulnerability detection, smart contracts, Ethereum blockchain

ACM Reference Format:

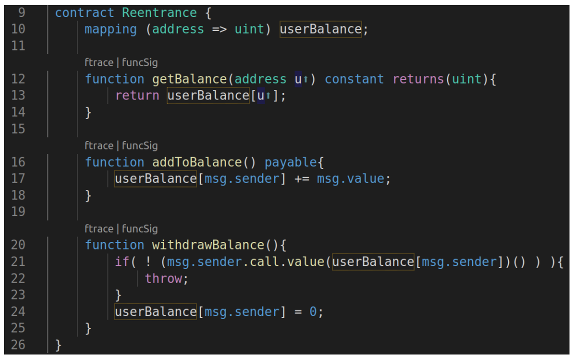
Anonymous Author(s). 2022. MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, August 14–18, 2022, Washington DC, USA.* , 11 pages.

1 INTRODUCTION

Graph learning has been an active research area for a long time. Learning *heterogeneous* graphs that consist of nodes and edges of different types has recently attracted extensive attention since such graphs contain richer information from the application domains than homogeneous graphs and, therefore, can achieve better learning results [37]. However, when it comes to complex heterogeneous graphs, where the graph structures have particular properties and the number of node types and edge types can be arbitrarily large and changing, it is still unclear if existing techniques can handle them well. Examples of such graphs can be found in control-flow graphs or call graphs representing possible software code execution flows and call relations.

This paper aims to develop a new approach for learning such complex and dynamic heterogeneous graphs and apply them to address critical software quality assurance problems, such as detecting vulnerabilities in software code that can be represented as control-flow graphs and call graphs. Expressly, we represent software code as a combination of heterogeneous graphs of multiple granularity levels that capture the syntactical control-flow and call relations in code. Then, we extract customized *metapaths* [6, 37] that acquire relations between different types of nodes and their neighbors and adjust various graph neural networks to learn the embeddings of the nodes and graphs at different granularity levels. Lastly, we train neural networks with embeddings to recognize an entire graph or a specific node that may contain vulnerabilities and eventually detect the vulnerable code lines or functions. We apply our approach to the Ethereum smart contracts written in the Solidity programming language. We choose smart contracts, running on distributed blockchains [17, 39], as they become increasingly popular in various domains that involve payments and contracts. Therefore, it is essential to provide diverse techniques to detect their potential bugs and ensure correct executions of the payments and contracts. We call our multi-level graph embeddings method for fine-grained smart contract vulnerabilities as MANDO. We create a mixed dataset containing 493 Solidity vulnerable contracts from multiple data sources from previous studies for our empirical evaluation. There are seven types of vulnerabilities in the dataset; each has between 50 to 80 instances. Our evaluation results show that MANDO achieves a heightened F1-score from 81.98% to 90.51% for detecting the vulnerabilities at the fine-grained line-level, while previous deep learning and embedding-based techniques can only detect the vulnerabilities at the contract file/function level. We also show that, compared to a few different graph embedding models (such as node2vec [13], LINE [31], GCN [15], and metapath2vec [6]) and traditional program analysis techniques (such as Securify [33], Mythril [26], Slither [8], Manticore [25], Smartcheck [32], and Oyente [24]) that can detect vulnerabilities at the line level, our method improves their F1-score by 11.35% to 70.81% for various bug types. The main contributions of this paper are as follows:

- We propose a new technique for representing Ethereum smart contracts written in Solidity as *heterogeneous contract graphs* that combines control-flow graphs (CFGs) and call graphs (CGs) of multiple levels of granularity;



```
9   contract Reentrance {
10      mapping (address => uint) userBalance;
11
        ftrace | funcSig
12      function getBalance(address u) constant returns(uint){
13          return userBalance[u];
14      }
15
        ftrace | funcSig
16      function addToBalance() payable{
17          userBalance[msg.sender] += msg.value;
18      }
19
        ftrace | funcSig
20      function withdrawBalance(){
21          if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
22              throw;
23          }
24          userBalance[msg.sender] = 0;
25      }
26  }
```

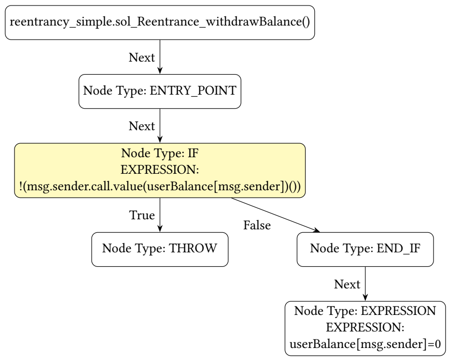


Figure 1: One example of an Ethereum smart contract code snippet (left) and its corresponding control-flow graph of the *withdrawBalance* function (right). The node containing a reentrancy bug has a yellow background.

- We tailor recent advances in heterogeneous attention graph neural networks [37] and metapaths [6] to build multi-level embeddings of heterogeneous graphs with multiple granularity levels;
- We employ the multi-level embeddings of heterogeneous graphs and labeled instances of vulnerable smart contracts to detect new vulnerabilities accurately at the line-level and contract-level, achieving better results than prior state-of-the-art bug detection techniques for smart contracts;
- We also publicize the dataset and our graph embedding models for the research community[1].

The rest of the paper is organized as follows. Section 2 defines our main research problem and objective with a motivating example. Section 3 describes the elements and structure of MANDO. Section 4 presents the experimental settings and results to show the effectiveness of our method. Section 5 reviews the related studies. Section 6 concludes our paper with some discussions on its limitations and future outlook.

2 MOTIVATION AND PROBLEM DEFINITION

Motivating Example: Figure 1 shows a sample code snippet of a smart contract written in Solidity. The flow chart on the right shows the corresponding control-flow and call graph of the sample contract. It contains a vulnerability at lines 21–24 as `userBalance` can be repeatedly sent to `msg.sender` at line 21 before it is set to 0 at line 24, which means `msg.sender` can receive more values than what is stored in `userBalance`. In order to catch this so-called *reentrance* vulnerability, the control-flow and call relations between `msg.sender` and `userBalance` should be considered. We aim to automatically capture such vulnerabilities' properties via our new graph embedding techniques.

Problem Statement: The high-level problem we aim to address is how to develop more effective heterogeneous graph learning techniques, which can be used to detect fine-grained line-level locations of software vulnerabilities and their types. More specifically, our objective for smart contracts written in Solidity based on our unique graph representation and embedding techniques is to: (1) Represent it as a *heterogeneous contract graph* that combines its control-flow graph and call graph like the example in Figure 1; (2) Learn the embeddings of the graphs and the nodes at multiple levels of granularity to capture the syntactical and semantic information of the contract code; (3) Efficiently and accurately identify the nodes that contain certain types of vulnerabilities and locate them in the contract code.

Usage Scenarios: Such efficient and accurate vulnerability detection can be useful for smart contract quality assurance under various situations. For example,

- During the contract development in an Integrated Development Environment (IDE), the vulnerability detection, if efficient enough with a negligible slowdown in IDE, can help the developer identify if the contract contains any known vulnerabilities type early.
- When a developer is reusing a contract from a third party, the vulnerability detection can check if it contains any known vulnerabilities and warns the developer about potential risks in reusing the contract directly.
- Whenever a new type of vulnerability is discovered, we may want to audit all existing contracts again to check if they contain the new type of vulnerability. The vulnerability detection can then be applied to all the contracts on a large scale for this purpose.

We also believe that MANDO can be generalized to other software code as long as their control-flow and call graphs can be constructed and there are known vulnerability datasets available for training. We will explore these applications in the future.

3 THE MANDO APPROACH

3.1 Overview

This section gives an overview of our proposed method that consists of four main components presented in the grey boxes in Figure 2 and describe each component in the following subsections.

As illustrated in Figure 2, the input of our proposed method is the source code of one or many Ethereum smart contract source files written in Solidity. The output is the bug prediction and the bug line in the source code if there is one.

First, the source code is processed by the **Heterogeneous Contract Graph Generator** component and translated into two heterogeneous graphs based on call graphs and control-flow graphs

[1]https://anonymous.4open.science/r/ge-sc-FE31

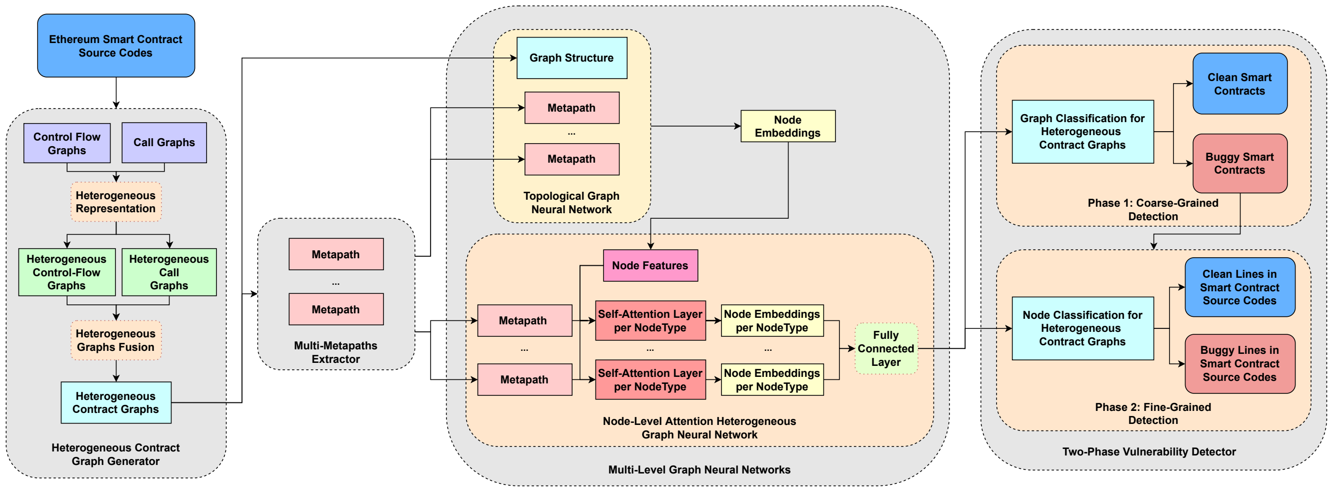

Ethereum Smart Contract Source Codes
Control Flow Graphs
Call Graphs
Heterogeneous Representation
Heterogeneous Control-Flow Graphs
Heterogeneous Call Graphs
Heterogeneous Graphs Fusion
Heterogeneous Contract Graphs
Heterogeneous Contract Graph Generator
Metapath
...
Metapath
Multi-Metapaths Extractor
Graph Structure
Metapath
...
Metapath
Topological Graph Neural Network
Node Embeddings
Node Features
Metapath
...
Metapath
Self-Attention Layer per NodeType
...
Self-Attention Layer per NodeType
Node Embeddings per NodeType
...
Node Embeddings per NodeType
Fully Connected Layer
Node-Level Attention Heterogeneous Graph Neural Network
Multi-Level Graph Neural Networks
Graph Classification for Heterogeneous Contract Graphs
Clean Smart Contracts
Buggy Smart Contracts
Phase 1: Coarse-Grained Detection
Node Classification for Heterogeneous Contract Graphs
Clean Lines in Smart Contract Source Codes
Buggy Lines in Smart Contract Source Codes
Phase 2: Fine-Grained Detection
Two-Phase Vulnerability Detector

Figure 2: Overview of the MANDO framework.

corresponding to two levels of granularity: contract level and statement (line) level, respectively. Then, the two heterogeneous graphs are fed into the second component: **Multi-Metapaths Extractor**. Based on the type of each node and the types of its associated edges, the component extracts their corresponding *metapaths*. This component is novel in the sense that it can handle dynamic node size and edge types in metapaths from the automatically generated heterogeneous contract graphs. The third component, **Multi-Level Graph Neural Networks**, contains two steps. The first step takes metapaths or graph topology of the contract graphs from the previous component as input and generates node embeddings. Then, in the second step, the node embeddings are used as node features and fused with metapaths using heterogeneous attention mechanisms at the node level. **Two-Phase Vulnerability Detector**, the last component, uses the embeddings to train fully connected layers to perform either graph classification or node classification, depending on the kind of the input heterogeneous contract graphs. In **Coarse-Grained Detection**, the heterogeneous contract graphs embeddings are used to classify graphs if their respective contract is clean or vulnerable. In **Fine-Grained Detection**, the heterogeneous contract graphs embeddings of the vulnerable contracts, classified in the first phase, are used to classify a node of a contract graph as to whether it is clean or vulnerable. The classified nodes can then be used to find the exact locations of the vulnerabilities in specific contracts (i.e., contract-level) and specific statements or lines of code (i.e., line-level).

3.2 Heterogeneous Contract Graph Generator

Definition 3.2.1 (Heterogeneous Graph). A heterogeneous graph is a directed graph $G = (V, E, \phi, \psi)$, consisting of a vertex set V and an edge set E. $\phi : V \longrightarrow A$ is a node-type mapping function and $\psi : E \longrightarrow R$ is an edge-type mapping function. A and R denote the sets of node types and edge types, and $|A| \geq 2$ and $|R| \geq 1$.

Our approach uses Slither [8] to traverse and analyze the source code of each Ethereum smart contract for generating the basic control-flow graphs and call graphs with homogeneous structures where nodes and edges have no types or labels. Then, we transform these constructed graphs into heterogeneous forms.

Heterogeneous Control-Flow Graphs (HCFGs). Control-flow graphs are an intermediate representation used in program analysis methods widely. These graphs depict all possible sequences of statements or lines of code that might be traversed during program executions. Recent approaches on smart contract vulnerability detection use such graph representations of code when applying graph neural networks [22, 42], but they mostly normalize and convert those representations into homogeneous graphs before applying graph models, and tend to lose valuable information regarding the semantic structures in the smart contract source code. In contrast, our approach focuses on retaining most of the structure and semantics of the source code through heterogeneous representations where a variety of node types and edge types are preserved, called *heterogeneous control-flow graphs*.

The set of all node types in control-flow graphs is denoted as A_{CF}. Some typical node types include ENTRY_POINT, EXPRESSION, NEW VARIABLE, RETURN, IF, END_IF, IF_LOOP, and END_LOOP. Additionally, diverse types of connections among nodes are used to describe statements' sequential or branching structure through edge types such as NEXT, IF_TRUE, IF_FALSE. The set of all edge types in control-flow graphs is R_{CF}. Figure 1 demonstrates a smart contract code snippet and the heterogeneous control-flow graph generated for its *withdrawBalance()* function. A Solidity parser (e.g., Slither) produces the complete sets of A_{CF} and R_{CF} based on the grammar of the Solidity language. $G_{CF} = \{V_{CF}, E_{CF}, \phi_{CF}, \psi_{CF}\}$ denotes an HCFG with V_{CF} and E_{CF} as its vertex and edge sets, respectively. Each node $i \in V_{CF}$ can be viewed as a tuple of (i, ϕ^i_{CF}), where i is the index of node and $\phi^i_{CF} \in A_{CF}$ is the type of node i. Similarly, each edge $(i, j) \in E_{CF}$ has an edge type $\psi^{i,j}_{CF} \in R_{CF}$. Each function in a smart contract can have an HCFG generated for it, and the HCFG has an entry node corresponding to the entry point/header of the function. A smart contract may be viewed as a set of HCFGs as it may contain more than one function.

Heterogeneous Call Graphs (HCGs). Call graphs are an intermediate representation of invocation relations among functions from the same smart contract or different smart contracts. A call graph generated via static program analysis often represents every possible call relation among functions in a program. Our study focuses on two major types of calls in smart contracts: *internal calls* for function calls inside one smart contract and *external calls* for function calls from a contract to others, represented by the two respective edge types INTERNAL_CALL and EXTERNAL_CALL. In addition, Solidity fallback functions are important in Ethereum blockchain, executed when a function identifier does not match any of the available functions in a smart contract or if no suitable data was provided for the function call. Many vulnerabilities in Ethereum smart contracts are directly or indirectly related to such fallback functions [4]. Therefore, we represent such fallback functions with a particular node type, called FALLBACK_NODE, besides the typical function node type FUNCTION_NODE.

One HCG is generated from each smart contract. $G_C = \{V_C, E_C, \phi_C, \psi_C\}$ denotes a heterogeneous call graph with V_C and E_C as its node and edge sets, respectively. Each node i in V_C can be viewed as a tuple (i, ϕ^i_C) where i is the index of node, $\phi^i_C \in A_C$ is the type of the node i and A_C is the set of all node types in G_C. Similarly, each edge $(i, j) \in E_C$ has an associate edge type $\psi^{i,j}_C \in R_C$.

Heterogeneous Contract Graphs: A Fusion Form of Heterogeneous Call Graphs and Heterogeneous Control-Flow Graphs. The structures of these two types of graphs for a smart contract can be shared or combined into a global graph to enrich information for learning. In MANDO, we design a core for HCGs and HCFGs fusion. Accordingly, the HCG edges of the smart contract act as bridges to link the discrete HCFGs of the smart contract functions into a global fused graph. Specifically, $G_{Fusion} = \{V_F, E_F, \phi_F, \psi_F\}$ denotes the fusion graph of the heterogeneous CG and the heterogeneous CFGs for a smart contract, where $V_F = V_C \cup V^1_{CF} \cup ... \cup V^N_{CF}$ and $E_F = E_C \cup E^1_{CF} \cup ... \cup E^N_{CF}$, and N is number of the heterogeneous CFGs for the contract. Intuitively, for each and every function node i in the call graph, the control-flow graph G^i_{CF} for the function is attached to the function node i at the entry node of G^i_{CF}, and thus the call graph is expanded with control-flow graphs to produce the heterogeneous contract graph.

3.3 Multi-Metapaths Extractor

Definition 3.3.1 (Metapath). A metapath θ is a path in the form of $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} ... \xrightarrow{R_l} A_{l+1}$, which defines a composite relation $R = R_1 \circ R_2 \circ ... \circ R_l$ between type A_1 and A_{l+1}, and $\circ$ denotes the composition operator on relations. Note that, the **length** of θ is the number of relations in θ.

Unlike graph datasets having simple heterogeneous structures such as DBIS [30] with only three node types, the node type size in our generated graphs is dynamic. It can reach sixteen node types with three different connection types per node type, especially in the heterogeneous control-flow graphs. Pre-defining all possible metapaths with any length according to all possible node types and edge types is a challenge due to the explosion of the possible association between each pair of node types with relevant edge types of a path, leading to increased data sparsity and reduced training accuracy. For example, in Figure 1, between a node of ENTRY_POINT type and a node of EXPRESSION type, several different node types can be included, such as IF and END_IF, and in other smart contracts, NEW_VARIABLE, IF_LOOP, and END_LOOP can also be included. Besides, the order of these node types can change dynamically, depending on the input smart contracts' structures.

In order to address the problem of exploding and changing metapaths, our method focuses on length-2 metapaths through reflective connections between adjacent nodes to extract multiple metapaths. For instance, the relation between two adjacent nodes of the types ENTRY_POINT and IF in Figure 1 can be described by a length-2 metapath: $ENTRY\_POINT \xrightarrow{next} IF \xrightarrow{back} ENTRY\_POINT$. HCFGs are mostly tree-like, having very few of their own back-edges induced by the LOOP-related statements in the source code. This can lead to the lack of metapaths connecting many leaf-node types in the graphs. Adding the "back" relations helps alleviate the lack and improves the completeness of the extracted metapaths.

Previous studies [30, 37] also used length-2 in their evaluation, and a length-N metapath can be decomposed into (N - 1) length-2 metapaths. Thus, we follow those studies by using length-2 to capture the unique semantic between each node types pair and their neighbors and leave longer metapaths for future evaluations. We extract the set of length-2 metapaths of each node types pair in a smart contract by following the same methods used in HAN [37].

3.4 Multi-Level Graph Neural Networks

This component has two major building blocks: *Topological Graph Neural Network* and *Node-Level Attention Heterogeneous Graph Neural Network*. The former learns an input graph topology, while the latter weights the importance of the metapaths in the graph.

3.4.1 *Topological Graph Neural Network.*

The main goal of this building block is to capture the graph topology without considering node or edge types. Each node i has a node embedding e_i such that e_i and the embedding vector e_j of the neighboring nodes j of i are near in the embedding space. Omitting node types are often reasonable for node search and identification since the connection topology among nodes is often sufficient in differentiating them from each other. For instance, in Figure 1, the nodes of types IF and END_IF are adjacent and have a causal relation since the END_IF is only executed when the condition within IF is false.

Various state-of-the-art neural network techniques can be used to generate node embeddings of graphs (see Section 4). Furthermore, we evaluate both homogeneous (e.g., node2vec) and heterogeneous (e.g., metapath2vec) graph neural networks to have a more comprehensive comparison in our empirical evaluations.

3.4.2 *Node-Level Attention Heterogeneous Graph Neural Network.*

There are two kinds of input sources for this building block: the node embeddings from the previous topological graph neural network and the metapaths from the Multi-Metapaths Extractor.

Node-Level Attention. Inspired by the node-level attention mechanism proposed by HAN [37], we also learn to weigh the importance of every metapath and node. The previous neural network produces a node embedding e_i for each node i; then, we construct a weighted node feature e'_i by the following linear transformation:

$$e'_i = W_{\phi_i} \cdot e_i, \quad (1)$$

where W_{ϕ_i} is the transformation matrix associated to the type ϕ_i of node i. Each node type ϕ_i has a specific matrix W_{ϕ_i} to increase the flexibility of the transformation by projecting each type into a separated weight space.

We measure the weight of a metapath Φ according to each node pair (i, j) by leveraging the self-attention mechanism [34] between i and j. The weight a_{ij}^{Φ} is defined as follows:

$$a_{ij}^{\Phi} = \text{softmax}_j(\text{ATT}([e'_i, e'_j]; \Phi)), \tag{2}$$

where ATT is a multi-layer perceptron [16] whose values of parameters are automatically learned through back-propagation. The input of such perceptron is the concatenation of two vectors e'_i and e'_j. We then normalize the output of ATT into the range between 0 and 1 by all neighbors of j in metapaths.

The metapath embedding m_i^{Φ} of a node i along a metapath Φ is a weighted sum of the node features of its neighbors with corresponding weights defined in Equation (2). The formula is as follows:

$$m_i^{\Phi} = \sigma\left(\sum_{j \in \mathcal{N}_i^{\Phi}} a_{ij}^{\Phi} \cdot e'_j\right), \tag{3}$$

where σ is the activation function, and $\mathcal{N}_i^{\Phi}$ denotes the neighbors of the node i according to the metapath Φ.

To overcome the obstacle of high variance of data in heterogeneous graphs, we borrow the idea of multi-head attention [35]. Particularly, the metapath embedding m_i^{Φ} of each node in Equation (3) is calculated K times and then concatenated to create a final embedding m'^{Φ}_i for each metapath.

3.4.3 Optimization for Detection. We employ the fully connected layer with the softmax function as an activation function for the graph and node classification tasks. The input of such a layer is dependent on the type of prediction tasks. In the graph classification task, the average of all node embeddings of the graph is used as input. In contrast, in the node classification task, we concatenate all node embeddings as a single vector and feed it to the layer. The loss function for the training process is cross-entropy, and the parameters of our model are learned through back-propagation.

3.5 Two-Phase Vulnerability Detector

This component has two main phases: *Coarse-Grained Detection* and *Fine-Grained Detection*. The first phase classifies clean versus vulnerable smart contracts at the coarse-grained contract level; the second phase identifies the actual locations of the vulnerabilities in the smart contract source code at the fine-grained line level. Our primary contribution is to provide line-level locations of the vulnerabilities using graph learning based techniques, while the previous graph learning based methods [23, 42] only report vulnerabilities at the contract or function level.

3.5.1 Phase 1: Coarse-Grained Detection. This phase classifies if a smart contract contains a vulnerability. We use the fused heterogeneous call graphs and control-flow graphs (i.e., heterogeneous contract graphs) and their embeddings to represent each input smart contract, and train the fully connected layer (Section 3.4.3) to predict clean or vulnerable contracts. As there can be many clean smart contracts, this classification assists in reducing the search space by filtering out those clean contracts and reducing noisy data before the second phase of fine-grained vulnerability detection at the line level.

3.5.2 Phase 2: Fine-Grained Detection. For the vulnerable smart contracts identified in the first phase, we apply node classification on the node embeddings of their Heterogeneous Contract Graphs to identify the nodes that may contain vulnerabilities, which correspond to statements or lines of code and allow us to detect the locations of the vulnerabilities at the fine-grained line level in smart contract source code.

4 EMPIRICAL EVALUATION

In this section, we present the experimental settings and evaluation results to answer these research questions: **RQ1:** The performance of our model compared to several state-of-the-art baselines on contract-level vulnerability classification, and **RQ2:** The performance of our model on line-level vulnerability detection.

4.1 Datasets

Our evaluation is carried out on a mixed dataset from three datasets:

Smartbugs Curated [7, 9] is a collection of vulnerable Ethereum smart contracts organized into nine types. This dataset is one of the most used real datasets for research in automated reasoning and testing of smart contracts written in Solidity. It contains 143 annotated contracts having 208 tagged vulnerabilities.

SolidiFI-Benchmark [12] is a synthetic dataset of vulnerable smart contracts. There are 9369 injected vulnerabilities in 350 distinct contracts, with seven different vulnerability types. To ensure consistency in the evaluation, we only focus on the seven types of vulnerabilities that are joint in both datasets, including: *Access Control*, *Arithmetic*, *Denial of Service*, *Front Running*, *Reentracy*, *Time Manipulation*, and *Unchecked Low Level Calls*.[2]

Clean Smart Contracts from Smartbugs Wild [7, 9] is a collection of 47,398 unique smart contracts from the Ethereum network. Based on the results of eleven integrated detection tools, the Smartbugs framework reports 2,742 contracts that do not contain any bugs, out of the 47,398 contracts. Thus, we use the 2,742 contracts as a set of clean contracts.

For the coarse-grained contract-level vulnerability classification tasks, we randomly take some smart contracts from the clean set and then mix them with the Smartbugs Curated and SolidiFi-Benchmark sets. We keep a ratio of 1:1 between clean and buggy contracts since this helps us create more balanced train/test sets for the tasks since there are only about 50 to 80 buggy contracts labeled per each bug type. For the fine-grained line-level vulnerability detection tasks, we use the dataset containing vulnerable smart contracts only, i.e., the union of SmartBugs Curated and SolidiFI-Benchmark sets. We do not use other datasets such as the ones of Zhuang *et al.* [42], Liu *et al.* [23] and eThor [28] because they do not have fine-grained line-level labels for the vulnerabilities.

Note that the Slither parser we use does not automatically generate the clean or vulnerable labels for a node. Instead, the nodes are labeled based on the lines of vulnerable code either manually

[2]Detailed description of each vulnerability type is provided in the supplementary material (Section D).

by Smartbugs authors or injected by the SolidiFI tool. For example, Line 21 in Figure 1 contains a reentrancy vulnerability labeled by Smartbugs; then, the yellow node in the Heterogeneous CFG is labeled as vulnerable.

4.2 Comparison Methods

4.2.1 Comparison to Graph-based neural network Methods: We use the four state-of-the-art methods, including: *node2vec* [13] learns node embeddings by minimizing the cross-entropy loss between the embedding of two nodes belonging to the same random walk with negative sampling; *LINE* [31] only differs from node2vec in the exact formulations of the loss functions and optimizing strategies. It focuses on learning vectors of closer nodes having either connection between them or sharing the same 1-hop neighborhood. Then, the final representation is formed by concatenating the two generated vectors; *Graph Convolutional Network* [15] generalizes the convolutional neural network to solve the node classification problem on graphs. Specifically, it uses the Laplacian matrix as a first-order approximation for the propagation among the layers of spectral graph convolutions; and *metapath2vec* [6] maximizes the likelihood of retaining the structures and semantics of the node/edge labels using the embedding of each node in heterogeneous graphs. Similar to node2vec, the negative sampling is obtained during the optimization process.

The output embeddings of the homogeneous and heterogeneous graph neural networks are used in two ways in our evaluation. First, we use them directly as the baselines for the coarse-grained graph classification tasks and fine-grained node classification tasks. Second, each of the graph neural networks is plugged into MANDO as the topological graph neural network. The generated embeddings are then considered as the node features fed to MANDO's Node-Level Attention Heterogeneous Graph Neural Network.

In addition, the one-hot vectors based on the Node-Type is also used as the node features, which allows MANDO to perform independently without relying on any added-in topological graph neural network.

Parameter Settings: The node embedding size is set to 128 for all models. We use adaptive learning rate [29] from 0.0005 to 0.01 in coarse-grained tasks and from 0.0002 to 0.005 in fine-grained tasks when training. For each GAT layer [35] of each metapath that feeds to the MANDO's Self-Attention Layer per Node Type, we set 8 multi-heads whose hidden size is 32. The numbers of learning epochs of coarse-grained and fine-grained tasks are 50 and 100, respectively, to reach converging. For node2vec, LINE, GCN, and metapath2vec, we use the authors' recommended settings to ensure the highest performance.

4.2.2 Comparison with Conventional Detection Tools. We also compare our method to six common smart contract vulnerability detection tools based on traditional software engineering approaches: *Manticore* [25] analyzes the symbolic execution of smart contracts and binaries; *Mythril* [26] uses symbolic execution, SMT solving, and taint analysis to find out the security vulnerabilities of smart contracts; *Oyente* [24] is an open-source tool to analyze symbolic execution to detect bugs in the Ethereum blockchain; *Securify* [33] can prove if the behavior of a smart contract is safe or not according to given predicates and by checking its graph dependencies; *Slither* [8] reduces the complexity of instruction sets with the intermediate representation of Ethereum smart contract called SlithIR, while retaining much of the semantic to increase the accuracy of bug detection; *Smartcheck* [32] converts smart contracts into XML-based representation and finds possible bugs along executive paths.

4.3 Evaluation Metrics

Since our prediction results are based on binary classification of a node or a graph, we use F1-score and Macro-F1 scores to measure the prediction performance. The former is a measure of a model's performance by balancing between precision and recall, while the latter is used to assess the quality of problems with multiple binary labels or multiple classes. In our evaluation, the F1-score metric is used to evaluate the models' performance when finding vulnerabilities in the graphs, and we also call it *Buggy-F1*. Macro-F1 is considered to avoid biases in the clean and vulnerability labels.

4.4 Empirical Results

In our evaluation, the average numbers of nodes and edges in Heterogeneous Contract Graphs are about 30,000 and 25,000 per vulnerability type, respectively. We randomly split the dataset to train/test sets with 70%/30% and maintained the ratio of vulnerable nodes in each set corresponding to the whole dataset. To get robust results for each dataset, each embedding method, and each vulnerability type, we run the experiment twenty times independently, each time with a different random seed, and then report the average results.

*4.4.1 Coarse-Grained Contract-Level Vulnerability Detection (**RQ1**).* In this experiment, we want to measure MANDO's performance with various node feature generator components in detecting vulnerable smart contracts (see Section 3.5.1). It illustrates the flexibility of our method working with different graph neural networks. Table 1 presents MANDO's performance via several different graph neural methods on various vulnerability types. Accordingly, we have some observations:

- MANDO generally outperforms baseline GNNs in contract-level detection. For instance, the Buggy-F1 and Macro-F1 of MANDO are over 88.66%, while the maximum performance of the baselines is 64.77% in detecting the Front-Running vulnerability type.
- It is unclear which node feature generation method is the best among the heterogeneous and homogeneous GNNs and the node-type one-hot vectors. However, integrating these types of GNNs inside MANDO outperforms all the baselines. Hence, we believe that the architecture of MANDO for combining different GNNs is suitable for classifying vulnerable smart contracts.
- MANDO is reliable in determining whether an unknown smart contract contains vulnerabilities, especially for the vulnerability types of Denial of Service, Front Running, and Time Manipulation with Buggy-F1 over 87.7%. MANDO is highly compatible with different solidity versions based on the Slither tool [8], and its trained models can be applied in practice to audit newly-appeared smart contracts that previous studies using graph learning [22, 42] have not been able to do effectively (see Section 5.1).

*4.4.2 Fine-Grained Line-Level Vulnerability Detection (**RQ2**).* To help smart contract developers to locate vulnerabilities more easily,

Methods		Metrics	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Heterogeneous GNN	metapath2vec	Buggy F1	62.90%	56.46%	55.17%	63.40%	61.79%	66.29%	55.22%
		Macro-F1	42.55%	46.32%	44.49%	43.03%	47.26%	45.94%	49.05%
Homogeneous GNNs	GCN	Buggy F1	60.63%	-	60.12%	-	-	59.60%	-
		Macro-F1	48.45%	-	45.65%	-	-	46.60%	-
	LINE	Buggy F1	61.45%	33.41%	59.61%	62.61%	66.23%	66.65%	60.51%
		Macro-F1	40.88%	33.47%	35.77%	34.29%	37.91%	40.84%	40.08%
	node2vec	Buggy F1	62.63%	58.59%	56.41%	64.77%	58.29%	63.03%	61.69%
		Macro-F1	48.83%	50.80%	40.63%	46.08%	45.80%	46.78%	49.91%
MANDO with Node Features Generated by	**NodeType One Hot Vectors**	Buggy F1	**71.19%**	**66.85%**	87.37%	87.31%	**76.09%**	85.03%	**72.08%**
		Macro-F1	**74.57%**	**71.04%**	86.68%	85.65%	**75.80%**	83.35%	**74.52%**
	metapath2vec	Buggy F1	57.70%	52.84%	60.16%	62.19%	55.06%	59.47%	51.37%
		Macro-F1	55.60%	55.06%	64.12%	64.80%	60.96%	57.74%	55.58%
	GCN	Buggy F1	49.26%	-	53.19%	-	-	49.50%	-
		Macro-F1	52.75%	-	60.26%	-	-	57.31%	-
	LINE	Buggy F1	65.12%	54.91%	**89.15%**	**89.86%**	71.04%	**87.71%**	59.44%
		Macro-F1	70.15%	65.36%	**89.46%**	**88.66%**	74.97%	**86.41%**	66.16%
	node2vec	Buggy F1	55.71%	64.11%	83.86%	86.05%	71.39%	73.38%	66.10%
		Macro-F1	64.70%	70.23%	83.40%	84.95%	72.31%	74.36%	71.02%

Table 1: Performance Comparison on the Coarse-Grained Contract-Level Detection. We use the *Heterogeneous Contract Graphs* of both Clean and Buggy Smart Contracts as the MANDO framework inputs. *Buggy- F1* means the F1-score of the buggy graph label. '–' denotes not applicable due to the insufficiency of GPU memory to handle the input graphs for the GCN model.

vulnerability detectors should be able to identify the vulnerabilities at the more fine-grained line level (see Section 3.5.2). In this experiment, we examine the performance of our method with respect to various state-of-the-art methods for line-level detection.

Table 2 shows the performance of our method trained with different models for Topological Graph Neural Network and the baselines methods, including graph-based neural networks and the conventional detection tools based on various software engineering techniques. From the table, we have the following observations:

- Generally, MANDO outperforms conventional detection tools significantly. Remarkably, an improvement is up to 63.4% of MANDO compared to the best performance of the tools in detecting Reentrancy bugs. We argue the significant improvement is from two sources: First, our constructed heterogeneous graphs retain more CFGs' aspects than other analysis tools. Secondly, our node-level attention module is flexible enough for GNNs to learn the exact locations of vulnerabilities within contracts.
- Our method beats the results of the baseline GNNs. Remarkably, the macro-F1 scores of the baseline GNNs are up to 60.5%, while our models can reach up to 80.78%. Hence, it is evident modeling the smart contracts as Heterogeneous Contract Graphs can benefit vulnerability prediction.
- Conventional detection tools perform well in detecting arithmetic bugs. The phenomenon is reasonable since these tools mostly use symbolic execution and such technique is suitable for detecting arithmetic bugs [2]. However, MANDO performance is still on par with the tools and our future work will improve the graph models to learn arithmetic operations better.

5 RELATED WORKS

5.1 Graph Embedding Neural Networks

A few studies have detected smart contract vulnerabilities using neural network-based embedding techniques. Zhuang *et al.* [42] represent each function's syntactic and semantic structures in smart contracts as a contract graph and propose a degree-free graph convolutional neural network with expert patterns to learn the normalized graphs for vulnerability detection. Their follow-up work [22] provides more interpretable weights in their graph neural network by extracting vulnerability-specific expert patterns for encoding graphs. In their Peculiar tool [40], Wu *et al.* present a pre-training technique based on customized data flow graphs of smart contract functions to identify reentrance vulnerabilities. However, their methods face various limitations: Relying on expert patterns, their graph generator only works with some pre-defined Major and Secondary functions before generating the contract graphs, leading to poor performance in the graph generation process compared to MANDO[3]. Besides, pre-defined patterns also restrict them to detect only two specified bugs, Reentrancy and Time Manipulation, in Solidity source code. In contrast, the heterogeneous graph structure allows MANDO to be more general and flexible in exploring many different vulnerability types without requiring any pre-definitions.

Other studies use other forms of embeddings: Zhao *et al.* [41] use word embedding together with similarity detection and Generative Adversarial Networks (GAN) to detect reentrance vulnerabilities dynamically. SmartConDetect [14] treats code fragments as unique sequences of tokens and uses a pre-trained BERT model to identify vulnerable patterns. SmartEmbed [10] employs serialized structured syntax trees to train word2vec and fastText models to recognize vulnerabilities. Different from such existing techniques, we propose to adopt multi-level heterogeneous attention graph neural networks to learn the contract graphs from fine-grained node-level to the whole graph-level. Our unique graph encodings can accurately capture the vulnerability patterns and locate fine-grained vulnerabilities at the line level.

5.2 Code Representation and Learning

Software programs have explored learning from heterogeneous graphs for vulnerability detection, code search, and other tasks [21]. For example, VulDeePecker [20] and SySeVR [19] use both syntax

[3]Section C in the supplementary material presents this in detail.

Methods		Metrics	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional Detection Tools	securify	Buggy F1	13.0%	0.0%	18.0%	53.0%	23.0%	24.0%	11.0%
		Macro-F1	52.3%	45.2%	52.0%	72.2%	58.4%	52.4%	54.1%
	mythril	Buggy F1	34.0%	73.0%	41.0%	63.0%	19.0%	23.0%	14.0%
		Macro-F1	61.1%	**84.1%**	60.1%	77.8%	55.3%	50.8%	55.7%
	slither	Buggy F1	32.0%	0.0%	13.0%	26.0%	15.0%	44.0%	10.0%
		Macro-F1	61.5%	45.2%	42.7%	56.9%	49.4%	57.3%	53.3%
	manticore	Buggy F1	30.0%	30.0%	12.0%	7.0%	9.0%	24.0%	4.0%
		Macro-F1	61.1%	61.0%	48.0%	46.9%	51.2%	55.1%	50.6%
	smartcheck	Buggy F1	20.0%	22.0%	52.0%	0.0%	22.0%	44.0%	11.0%
		Macro-F1	56.0%	56.1%	69.9%	46.2%	57.8%	64.2%	54.1%
	oyente	Buggy F1	21.0%	71.0%	48.0%	0.0%	20.0%	24.0%	8.0%
		Macro-F1	57.3%	82.8%	67.2%	44.8%	56.1%	52.4%	52.6%
Heterogeneous GNN	metapath2vec	Buggy F1	35.46%	68.70%	60.64%	80.65%	71.66%	67.51%	26.06%
		Macro-F1	48.52%	47.08%	48.67%	49.88%	49.15%	49.00%	49.91%
Homogeneous GNNs	GCN	Buggy F1	43.92%	65.69%	64.06%	81.09%	71.76%	68.70%	38.13%
		Macro-F1	54.20%	53.42%	54.81%	56.21%	53.00%	52.74%	53.57%
	LINE	Buggy F1	53.59%	68.61%	62.28%	83.06%	74.78%	70.76%	7.10%
		Macro-F1	57.75%	48.53%	51.63%	42.27%	38.26%	42.40%	44.31%
	node2vec	Buggy F1	44.94%	67.84%	63.92%	81.84%	71.52%	67.81%	34.26%
		Macro-F1	54.73%	52.92%	54.83%	56.17%	53.45%	53.19%	53.09%
MANDO with Node Features Generated by	**NodeType One Hot Vectors**	Buggy F1	77.21%	81.62%	79.83%	88.19%	84.24%	86.64%	65.95%
		Macro-F1	74.89%	76.01%	76.22%	68.70%	75.89%	82.72%	75.01%
	metapath2vec	Buggy F1	67.97%	74.84%	67.22%	86.08%	76.03%	73.81%	50.71%
		Macro-F1	67.87%	65.92%	62.90%	65.22%	66.04%	71.04%	64.73%
	GCN	Buggy F1	69.00%	76.47%	70.88%	87.15%	77.57%	77.73%	52.95%
		Macro-F1	66.77%	66.75%	64.26%	65.71%	65.85%	73.94%	65.75%
	LINE	Buggy F1	81.19%	81.58%	**82.12%**	90.47%	86.27%	89.21%	83.37%
		Macro-F1	**80.93%**	77.80%	**79.00%**	78.43%	80.43%	86.17%	85.40%
	node2vec	Buggy F1	**81.98%**	**84.35%**	82.09%	**90.51%**	**86.40%**	**90.29%**	**84.81%**
		Macro-F1	79.23%	79.10%	77.84%	**78.60%**	**80.78%**	**86.76%**	**86.74%**

Table 2: Performance Comparison on the Fine-Grained Line-Level Detection. We use the *Heterogeneous Contract Graphs* of the Buggy Smart Contracts as the inputs for MANDO framework. *Buggy- F1* means the F1-score of the buggy node label. A total of fifteen methods are examined in the comparisons. The best performance in each vulnerability category is highlighted.

structures and dependency slices to represent programs and employ commonly used neural network models to learn the programs' embedding and identify vulnerability patterns for C/C++ programs. VulDeeLocator [18] extends the work by adding attention-based granularity refinement to identify fine-grained line-level vulnerability locations. BGNN4VD [3] also uses combined code representations in the abstract syntax trees and control- and data-flow graphs to learn vulnerability patterns via bilateral graph neural networks for C/C++ programs. However, no such study has been done for Solidity smart contracts and our study is the first to consider heterogeneous graph learning that combines control-flow graphs and call graphs together and employs the learned embedding for recognizing vulnerabilities in Solidity smart contracts.

5.3 Bug Detection & Smart Contracts

Smart contracts in blockchain technology have received increasing popularity and reputation, and hence, vulnerability detection for smart contracts is becoming a popular topic [1, 5]. Several studies detect specific types of bugs or vulnerabilities using traditional program analysis and software engineering and security techniques. For example, OYENTE [24] uses symbolic execution to explore execution paths in smart contracts as much as possible and search for four types of bugs. SmartCheck [32] uses static analysis techniques to check smart contract code for patterns that match pre-defined rules about vulnerabilities and code smells. Several other studies use formal verification to check smart contracts' safety and functional correctness according to certain human-defined specifications [11]. In addition to symbolic execution and formal verification, many studies on smart contract security are based on abstract interpretation, fuzz testing, enhanced compilation, dynamic consistency checking, and other techniques [4, 27, 36, 38]. Such security analysis techniques are built to discover specific vulnerabilities according to manually defined patterns or specifications, limiting their scalability and accuracy. Our proposed method is entirely different since we automatically recognize bug patterns using neural networks.

6 CONCLUSION AND FUTURE WORK

The popularity and importance of smart contracts in blockchain platforms are increasing. Therefore, it is highly desirable to ensure the quality and security of smart contract programs. In this paper, we propose a new method, based on multi-level graph embeddings of control-flow graphs and call graphs of Solidity smart contracts, to train more accurate vulnerability detection models that can identify vulnerabilities in smart contracts at fine-grained line level and contract level of granularity. Our evaluation on a large-scale dataset curated from real-world Solidity smart contracts shows that our method is promising and outperforms several baselines: six conventional detection tools and four state-of-the-art graph neural networks. Our method is thus a valuable complement to other vulnerability detection techniques and contributes to smart contract security. However, with all the achievements and progress, our method and evaluation can still be improved further. The embedding techniques can further fuse more semantic properties of the smart contract source code, such as data dependencies, and adapt newer and more sophisticated graph neural networks. We can also adapt our method to cases where only compiled smart contract bytecode is available without source code to expand the applicable scope and perform even larger-scale evaluation. The evaluation can further compare and contrast with vulnerability detection techniques developed for other programming languages (e.g., C/C++, Java) to check the generalizability of our method. We leave all the many exciting research problems for near future work.

REFERENCES

[1] M. Alharby, A. Aldweesh, and A. v. Moorsel. 2018. Blockchain-based Smart Contracts: A Systematic Mapping Study of Academic Research. In *International Conference on Cloud Computing, Big Data and Blockchain*. 1–6.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018).

[3] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology* 136 (2021), 106576.

[4] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.

[5] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*. 79–94.

[6] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 135–144.

[7] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *the ACM/IEEE 42nd International Conference on Software Engineering*. 530–541.

[8] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. 8–15.

[9] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: a framework to analyze solidity smart contracts. In *the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352.

[10] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering* (2020).

[11] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. 2021. A Survey on Formal Verification for Solidity Smart Contracts. In *2021 Australasian Computer Science Week Multiconference*. 1–10.

[12] Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[13] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 855–864.

[14] Sowon Jeon, Gilhee Lee, Hyoungshick Kim, and Simon S Woo. 2021. SmartConDetect: Highly Accurate Smart Contract Code Vulnerability Detection Mechanism using BERT. In *KDD Workshop on Programming Language Processing*.

[15] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[17] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2017. A survey on the security of blockchain systems. (2017), 25 pages.

[18] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. VulDeeLocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* (2021).

[19] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[20] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *The Network and Distributed System Security Symposium*.

[21] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.

[22] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. *arXiv preprint arXiv:2106.09282* (2021).

[23] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *the ACM SIGSAC conference on computer and communications security*. 254–269.

[25] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *the 34th IEEE/ACM International Conference on Automated Software Engineering*. 1186–1189.

[26] Bernhard Mueller. [n. d.]. Smashing Smart Contracts for Fun and Real Profit. In *9th annual HITB Security Conference* (2018). 2–51.

[27] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[28] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 621–640.

[29] Leslie N Smith and Nicholay Topin. 2019. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial intelligence and machine learning for multi-domain operations applications*, Vol. 11006. International Society for Optics and Photonics, 1100612.

[30] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *the VLDB Endowment* 4, 11 (2011), 992–1003.

[31] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW*.

[32] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[33] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *25th ACM Conference on Computer and Communications Security*.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[35] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[36] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. VULTRON: catching vulnerable smart contracts once and for all. In *IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results*. 1–4.

[37] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous graph attention network. In *The World Wide Web Conference*. 2022–2032.

[38] Yajing Wang, Jingsha He, Nafei Zhu, Yuzi Yi, Qingqing Zhang, Hongyu Song, and Ruixin Xue. 2021. Security enhancement technologies for smart contracts in the blockchain: A survey. *Transactions on Emerging Telecommunications Technologies* (2021).

[39] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[40] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques. In *the 32nd International Symposium on Software Reliability Engineering*.

[41] Hui Zhao, Peng Su, Yihang Wei, Keke Gai, and Meikang Qiu. 2021. GAN-Enabled Code Embedding for Reentrant Vulnerabilities Detection. In *Knowledge Science, Engineering and Management*. 585–597.

[42] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network.. In *IJCAI*. 3283–3290.

Supplemental Material

A ARTIFACT AVAILABILITY

We will release the dataset and our graph embedding models used in the study for the research community. For the review, it is available at this anonymous link: https://anonymous.4open.science/r/ge-sc-FE31.

B EXTENDED EVALUATION

B.1 Coarse-Grained Contract-Level Detection with Inputs Being Only Heterogeneous Control-Flow Graphs

We examine this extended experiment with a similar dataset for each type of bug, presented in Section 4.4.2 of our main paper. The difference between the two experiments being the input of MANDO is the Heterogeneous Control-Flow Graphs only instead of the Heterogeneous Contract Graphs fused by HCGs and HCFGs. Interestingly, compared the methods achieving the best performance of Table 2 in the main paper to the best ones of Table 3, we realize that with the inputs from only HCFGs, the Buggy F1 and Macro-F1 of MANDO increase slightly about 1.57% to 9.17% with the bugs Time Manipulation and Arithmetic. This problem regards the characteristics of these two vulnerabilities that only depend on the statements inside each function of smart contracts, not the invocation of functions. Therefore, attaching the edge information of HCGs to HCFGs as the fusion form in Heterogenous Contract Graphs may noise the predictions of graph classification tasks. However, MANDO always beat the baseline graph neural networks with a significant gap in detecting smart contracts with bugs, whether inputs of HCFGs-only or the Fused Heterogeneous Contract Graphs.

B.2 Fine-Grained Function-Level Detection of Smart Contract Vulnerabilities

In the extended experiment, we propose another fusion form of Heterogeneous Call Graphs and Heterogeneous Control-Flow Graphs, named Form-B Fusion. Different from the core fusion form where the HCFGs are attached to the HCG, each HCFG for a function in the smart contract in Form-B is considered as additional features for the function node in the HCG, and these features are generated by the component "Topological Graph Neural Network" in MANDO, which we describe in Section 3.4. Table 4 shows that our proposed method achieves significant improvements up to 35% compared to the original metapath2vec method; moreover, it improves F1-score of the predicted bugs slightly by 1% to 7% compared to the original node2vec model in all the bug categories.

C PERFORMANCE OF GRAPH GENERATION PROCESSES

The graph extraction tools of Zhuang and Liu et al. [22, 42] have two internal steps. The first step extracts source code from the smart contracts to their pre-defined graph formats. Furthermore, in the second step, they vectorize these graphs to feed their graph neural networks. Notably, each type of bug requires a different tool for extraction and vectorization since it relies on some expert patterns regarding each bug type. This is why their approaches only support two types of vulnerability, including Reentrancy and Time Manipulation, and cannot expand to other bug types. In contrast, the graph generator of the MANDO framework is an entirely-automatic approach, which does not need any pre-defined patterns. Besides, our generated heterogeneous graphs are sufficiently general to capture the semantic structure of many different vulnerability types: Access Control, Arithmetic, Denial of Service, Front Running, Reentrancy, Time Manipulation, and Unchecked Low-Level Calls.

We utilize the Smartbugs Wild dataset having 47,398 smart contracts to compare MANDO with their graph extraction tools. In the Reentrancy bug, Zhuang and Liu et al. tools could extract 47279 (99.75%) contracts but only successfully vectorize 1524 contracts from that and get a total of 3.22% performance. In the Time Manipulation bug, they could extract 47376 (99.95%) contracts and also only vectorize 1151 contracts successfully from that and get a total of 2,43% performance. In the meantime, MANDO's graph generator and metapath extractor can successfully generate heterogeneous contracts and extract metapaths on 44331/47398 smart contracts for all bug categories before feeding to the multi-level graph neural network, getting 93.53% in the overall performance.

D DESCRIPTION OF VULNERABILITIES IN SOLIDITY SOURCE CODE

In this section, we would shortly describe some bug categories collected by Smartbugs or injected by SolidiFI, including:

- *Access Control*: Failure to use function modifiers or use of tx.origin.
- *Arithmetic*: Bugs related to the overflow or underflow of integer.
- *Denial of Service*: time-consuming operation leads to the rejection of the smart contract.
- *Front Running*: Two dependent transactions that invoke the same contract are included in one block.
- *Reentracy*: Unexpected behavior of a contract due to reentrant function calls.
- *Time manipulation*: The timestamp of the block is manipulated by the miner.
- *Unchecked Low Level Calls*: low level functions i.e. call(), callcode(), delegatecall() or send() fails because of unchecked condition.

Methods		Metrics	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Original Heterogeneous GNN	metapath2vec	Buggy F1	38.40%	47.36%	39.93%	26.84%	32.84%	32.68%	41.63%
		Macro-F1	41.11%	44.73%	38.27%	33.46%	43.02%	40.98%	43.49%
Original Homogeneous GNNs	GCN	Buggy F1	39.98%	-	45.25%	-	-	36.66%	-
		Macro-F1	38.87%	-	37.79%	-	-	41.91%	-
	LINE	Buggy F1	39.10%	42.20%	51.21%	42.14%	53.00%	43.45%	52.26%
		Macro-F1	33.43%	33.12%	32.18%	29.11%	36.59%	33.09%	33.91%
	node2vec	Buggy F1	39.56%	43.67%	53.43%	47.84%	57.04%	45.62%	50.21%
		Macro-F1	38.52%	37.33%	39.77%	35.73%	44.08%	37.24%	46.58%
***HCFGs only.* MANDO with Node Features Generated by**	**NodeType One Hot Vectors**	Buggy F1	**69.99%**	**76.04%**	83.85%	84.04%	**70.20%**	**89.35%**	**69.61%**
		Macro-F1	**69.23%**	**77.24%**	82.67%	80.37%	**72.91%**	**88.48%**	**70.86%**
	metapath2vec	Buggy F1	53.99%	53.94%	59.14%	61.34%	52.26%	56.55%	49.69%
		Macro-F1	55.97%	56.82%	61.12%	64.54%	58.53%	58.65%	54.75%
	GCN	Buggy F1	45.31%	-	55.71%	-	-	50.17%	-
		Macro-F1	49.10%	-	63.41%	-	-	58.09%	-
	LINE	Buggy F1	64.10%	60.46%	**85.22%**	**87.23%**	44.40%	87.54%	54.84%
		Macro-F1	66.47%	67.65%	**84.36%**	**84.37%**	58.83%	87.80%	62.73%
	node2vec	Buggy F1	59.55%	60.42%	81.50%	83.77%	56.69%	75.29%	56.17%
		Macro-F1	66.96%	67.55%	80.03%	80.01%	66.47%	76.44%	64.32%

Table 3: Performance Comparison on the Coarse-Grained Contract-Level Detection. We use the *Heterogeneous Control-Flow Graphs* of both Clean and Buggy Smart Contracts as the inputs for MANDO framework. *Buggy- F1* means the F1-score of the buggy graph label. The best performance in each vulnerability category is highlighted. '–' denotes not applicable due to the insufficiency of GPU memory to handle the input graphs for GCN model.

Methods		Metric	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Original Heterogeneous GNN	metapath2vec	Buggy F1	48.0%	62.0%	58.0%	69.0%	49.0%	51.0%	47.0%
Original Homogeneous GNN	GCN	Buggy F1	56%	69%	64%	77%	54%	61%	53%
	LINE	Buggy F1	75.0%	79.0%	79.0%	87.0%	70.0%	74.0%	47.0%
	node2vec	Buggy F1	81.0%	87.0%	85.0%	89.0%	81.0%	83.0%	77.0%
***Fusion (Form-B) of Call Graphs and Control-Flow Graphs.* MANDO with Node Features Generated by**	**NodeType One Hot Vectors**	Buggy F1	**83.0%**	87.0%	84.0%	86.0%	80.0%	**85.0%**	82.0%
	metapath2vec	Buggy F1	82.0%	**87.0%**	81.0%	89.0%	81.0%	85.0%	77.0%
	GCN	Buggy F1	81.0%	85.0%	82.0%	89.0%	80.0%	84.0%	82.0%
	LINE	Buggy F1	81.0%	86.0%	83.0%	**91.0%**	**82.0%**	81.0%	80.0%
	node2vec	Buggy F1	82.0%	87.0%	**87.0%**	91.0%	80.0%	84.0%	**84.0%**

Table 4: Performance Comparison on the Fine-Grained Function-Level Detection. We use *Heterogeneous Call Graphs* of the Buggy Smart Contracts as the inputs for MANDO framework, with *Heterogeneous Control-Flow Graphs* as the additional features of the function nodes of the input graphs. *Buggy F1* means the F1-score of the buggy function-node label. The best performance in each vulnerability category is highlighted.