MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection

Hoang H. Nguyen*
L3S Research Center
Leibniz Universität Hannover
Hannover, Germany
ehoang@l3s.de

Nhat-Minh Nguyen*
Singapore Management University
Singapore
nmnguyen@smu.edu.sg

Chunyao Xie
L3S Research Center
Leibniz Universität Hannover
Hannover, Germany
xie@l3s.de

Zahra Ahmadi
L3S Research Center
Leibniz Universität Hannover
Hannover, Germany
ahmadi@l3s.de

Daniel Kudendo
L3S Research Center
Leibniz Universität Hannover
Hannover, Germany
kudenko@l3s.de

Thanh-Nam Doan
Independent Researcher
Atlanta, Georgia, USA
me@tndoan.com

Lingxiao Jiang
Singapore Management University
Singapore
lxjiang@smu.edu.sg

***Abstract*—Smart contracts in blockchains have been increasingly used for high-value business applications. It is essential to check smart contracts' reliability before and after deployment. Although various program analysis and deep learning techniques have been proposed to detect vulnerabilities in either Ethereum smart contract source code or bytecode, their detection accuracy and scalability are still limited. This paper presents a novel framework named MANDO-HGT for detecting smart contract vulnerabilities. Given Ethereum smart contracts, either in source code or bytecode form, and vulnerable or clean, MANDO-HGT custom-builds heterogeneous contract graphs (HCGs) to represent control-flow and/or function-call information of the code. It then adapts heterogeneous graph transformers (HGTs) with customized meta relations for graph nodes and edges to learn their embeddings and train classifiers for detecting various vulnerability types in the nodes and graphs of the contracts more accurately. We have collected more than 55K Ethereum smart contracts from various data sources and verified the labels for 423 buggy and 2,742 clean contracts to evaluate MANDO-HGT. Our empirical results show that MANDO-HGT can significantly improve the detection accuracy of other state-of-the-art vulnerability detection techniques that are based on either machine learning or conventional analysis techniques. The accuracy improvements in terms of F1-score range from 0.7% to more than 76% at either the coarse-grained contract level or the fine-grained line level for various vulnerability types in either source code or bytecode. Our method is general and can be retrained easily for different vulnerability types without the need for manually defined vulnerability patterns.**

***Index Terms*—vulnerability detection, smart contracts, source code, bytecode, heterogeneous graph learning, graph transformer**

Acknowledgments. This work was supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 833635 (project ROXANNE: Real-time network, text, and speaker analytics for combating organized crime, 2019-2022) and by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant and the Lee Kong Chian Fellowship. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not reflect the views of any of the grantors. We also thank all the anonymous reviewers for their insightful feedback on our paper.

* The first author and the second author contributed equally to this work. H.H.N. conceived the original idea and experimental settings and directed the project. H.H.N. and N.M.N. developed the framework. N.M.N. performed the experiments for the evaluation.

I. Introduction

Smart contracts on blockchain systems have been used for many application domains [1], such as finance, e-commerce, healthcare, logistics, and law. Any bug or security vulnerability in a smart contract deployed in a blockchain can have devastating consequences for both the developers and the users of the smart contracts [2], [3]. Therefore, there is a high demand for various kinds of security assurance techniques for smart contracts, especially for vulnerability detection.

Many studies have been carried out on vulnerability detection in smart contracts based on conventional software testing, analysis, verification techniques [4]–[11]. Such techniques often require certain types of oracles or specifications of the (un)expected patterns or semantics of smart contract code for analysis. Unfortunately, specifying the patterns can take much manual effort and make it troublesome to adapt the tools to the evolving contract languages and types of vulnerabilities. Also, computational complexity makes it very expensive to repeatedly run the techniques on a large set of smart contracts to search for new types of vulnerabilities. Hence, a new class of vulnerability detection techniques has been proposed based on machine learning and deep learning techniques [12]–[17]. Such techniques aim to encode various syntactic and semantic code information via syntax trees, control-flow graphs, or program dependency graphs, among others, and to train automated classifiers to distinguish vulnerable code from normal ones. The learning-based techniques reduce the need for manually specified patterns or specifications, easier to be adapted to new types of code and vulnerabilities as long as some training data is provided. However, existing code learning techniques often treat most nodes and edges *homogeneously*, ignoring fine-grained differences in the nodes and edges types and their exact locations in the code's trees

and graphs. This leads to insufficient learning accuracy, and this limitation becomes more pronounced for smart contracts bytecode without source code. Since the structures representing different bytecode become more similar to each other if the types of the specific bytecode instructions are ignored, making it harder to identify vulnerable/bug patterns [1].

Combining the advantages of previous techniques and progresses in heterogeneous graph learning [18]–[21], this paper aims to develop a new framework for smart contract vulnerability detection, applicable to both source code and bytecode. The main idea of our framework is two-folded:

- First, we represent the contract code, either source code or bytecode, as customized heterogeneous contract graphs (HCGs) that represent control flows and call relations of the code. With the combined representations, we aim to capture the code's syntactic and semantic information more comprehensively to facilitate learning of code patterns and distinguishing vulnerable code from clean ones.
- Second, we extend the heterogeneous graph transformer (HGT) [21] techniques to learn different types of nodes and edges of the contract graphs and encode the semantics of the code more accurately. The encodings can then be used to train classifiers to recognize vulnerable code.

We name our framework MANDO-HGT, following a previous work [22] that only works for source code and uses a different graph learning technique. Our framework aims to be more general than previous techniques, applicable for either source code or bytecode, can be instantiated with various graph learning techniques, and can be re-trained for new types of vulnerabilities and detect vulnerabilities in large sets of contracts efficiently and accurately. The general approach is useful since it is not uncommon for smart contracts to be deployed *without* their source code or for the source code to be lost or deleted over time and the approach should be able to handle variation in generated bytecode to some extent when retrained with bytecode variants.

We have curated 55k Ethereum smart contracts from various data sources, including SmartBugs [23], [24] and SolidiFI-Benchmark [25], then verified the labels for 423 buggy and 2,742 clean contracts and evaluated MANDO-HGT on the mixed dataset. Our evaluation shows that MANDO-HGT significantly improves F1-score over other vulnerability detection techniques: (1) Compared to other best-performing learning-based techniques, it improves their F1-score by 0.74% to 22.56% at the contract level and 3.51% to 7.48% at a more fine-grained line level for various vulnerability types in either source code or bytecode; (2) Compared to best-performing conventional analysis-based techniques that detect vulnerabilities at the fine-grained line level, it improves their F1-score by 18.18% to 76.89%; Furthermore, MANDO-HGT can be re-trained to detect different types of vulnerabilities without manually defining bug patterns needed by analysis-based techniques.

We also show that, through a few case studies assisted by recent neural network interpretation techniques [26], [27], the detection results of MANDO-HGT are often meaningful, reflecting our understanding of the bug patterns. We also provide possible explanations for the failures in a few cases where the detection results are wrong, which may guide future improvements to learning-based techniques.

The rest of the paper is organized as follows. Section II provides a motivating example to show the benefits of our method. using heterogeneous graph transformers together with heterogeneous contract graphs for smart contracts. Section III briefly discusses our differences from related studies. Section IV provides more details of MANDO-HGT. Section V presents our evaluation results and a few case studies on their interpretations. Section VI concludes with future work.

II. Motivation

Motivating Sample Source Code. Figure 1 (Part A) presents a snippet of a smart contract written in Solidity with a *reentrance* vulnerability. Part B presents the call graph (CG) of the contract, and part C presents a partial sample control-flow graph (CFG) for the `collect` function of the sample contract. Line 11. `msg.sender.call`, is the root cause of the vulnerability of this sample code. `collect` can be repeatedly called before `balances` is deducted at Line 12, allowing `msg.sender` to receive more values than what is specified by `_am`. In order to catch this so-called *reentrance* vulnerability, the control-flow and call relations among `msg.sender`, `balances`, and `_am` should be considered.

Motivating Sample Bytecode. Figure 2 shows a snippet of a smart contract written in Solidity, together with its runtime bytecode [2] and the control-flow graph of the bytecode. It contains an *access control* vulnerability on lines 3–4 as `selfdestruct` is a critical function in Ethereum, leading to the self-destruction of the smart contract but inadequately protected. Thus, malicious parties can destruct the contract due to missing access controls. This vulnerability would be represented by two node-edge-type relations in our heterogeneous control-flow graph as $DISPATCHER \xrightarrow{true} DISPATCHER$ and $DISPATCHER \xrightarrow{next} LEAF$.

Objectives. Our primary objective is to automatically capture vulnerabilities in either contract source code like Figure 1 and contract bytecode like Figure 2 via our graph embedding techniques. More specifically, our objective is to: (1) Represent the Solidity source code or EVM bytecode as heterogeneous call and control-flow graphs like the example flow charts; (2) Learn the embeddings of the nodes and graphs; and (3) Efficiently and accurately identify if a contract contains vulnerabilities and locate them if its source code is available.

[1]In cybersecurity contexts, vulnerabilities mean special kinds of bugs that can be exploited and cause major security concerns. According to https://dasp.co/, all the bug types mentioned in this work can be vulnerabilities, although indeed, only a few instances of the bug types can be exploited. In this paper, we do not need to handle the differentiation and simply treat them as synonyms.

[2]When a contract in Solidity source code is compiled, the produced bytecode has two types: *creation* bytecode is the constructor code of the contract that performs initializations and deploys the *runtime* bytecode to the blockchain; the constructor code is then discarded, not stored in blockchain.

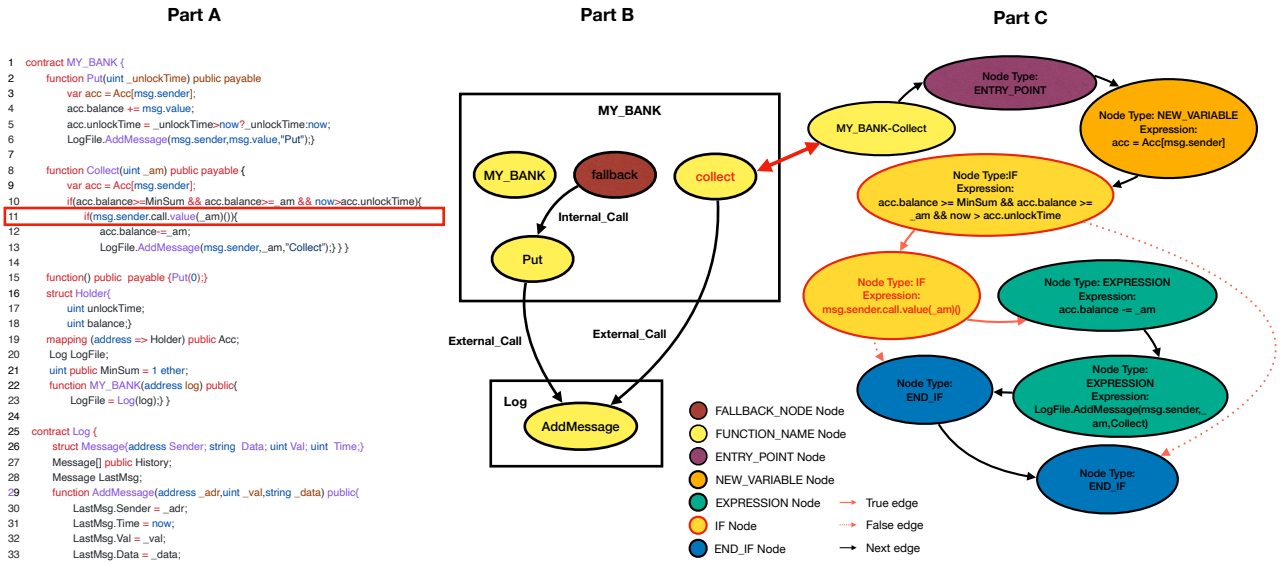

Fig. 1. A sample Ethereum smart contract `MY_BANK` (Part A), its call graph (CG) (Part B), and a control-flow graph (CFG) (Part C) for the function `Collect`. Line 11 in Part A is the root cause of a *reentrancy* bug; the nodes in CG and CFG containing the *reentrancy* bug are highlighted with red text.

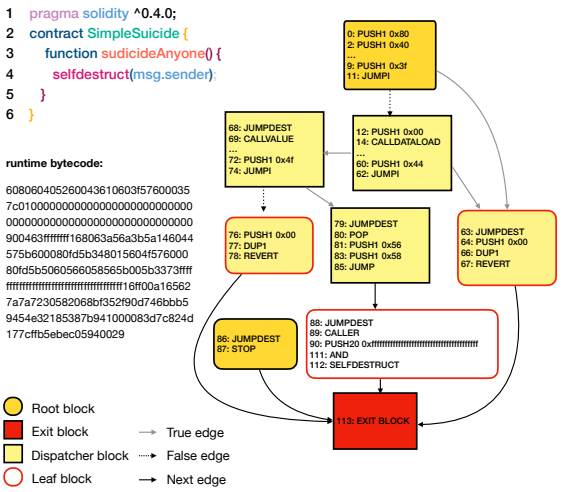

Fig. 2. Code snippet, runtime bytecode, and control-flow graph of the runtime bytecode of a contract containing an *access control* bug.

III. Related Work

We discuss the main differences between our work and closely related work on smart contract bug detection.

Conventional bug detection techniques. Many studies on vulnerability detection are based on conventional techniques, such as testing/fuzzing [4], [28]–[31], symbolic execution [5], [6], [32]–[35], static/dynamic program analysis [7]–[9], [36]–[39], and formal verification [10], [11], [40]–[44]. They often need customized implementation of the testing, analysis, and verification algorithms for the specific smart contract language and vulnerability types; their analysis algorithms can be very different for source code and bytecode, limiting their flexibility for new languages or vulnerability types. Although our learning-based approach also requires customized front-end code parsing and control-flow graph constructions, the graph-learning components are independent of the languages and can be applicable to new vulnerability types.

Learning-based bug detection techniques. There are also many studies based on machine learning and deep learning for detecting bugs in either source, bytecode, or binary. Some studies consider different kinds of code representations and learning techniques for source code in various languages [45]–[53], but very few consider heterogeneous graph learning. A recent study uses heterogeneous graphs for source code [54], but it has not yet been applied to control-flow graphs of smart contracts. Also, among the limited literature on deep learning-based vulnerability detection methods in smart contracts bytecode [13], [16], [55]–[60], some use control-flow and data-flow graphs (and bytecode instruction sequences). However, they still use homogeneous graph learning techniques, while our approach customizes heterogeneous graph learning for both source code and bytecode of smart contracts.

There are other general code representation learning studies [61]–[66] that are different from ours but can potentially be combined with ours to improve its accuracy in encoding bugs. We leave such interesting exploration for future work.

IV. Approach

Our proposed framework, MANDO-HGT, consists of five main components in the grey boxes in Figure 3: *Heterogeneous Contract Graph Generator*, *Meta Relations Extractor*, *Node Features Extractor*, *MANDO-HGT Graph Neural Network*, and *Two-Phase Vulnerability Detector*. The five components are explained in detail below. The input of MANDO-HGT is either the source code or bytecode of one or more Ethereum smart contracts, and the output is the bug predictions for the input contracts at the contract level (for both source code and bytecode) and the line-level (for source code only).

A. Heterogeneous Contract Graph Generator

Definition IV-A.1 (Heterogeneous Graph)**.** A heterogeneous graph is a directed graph $G = (V, E, \tau, \phi)$, consisting of a

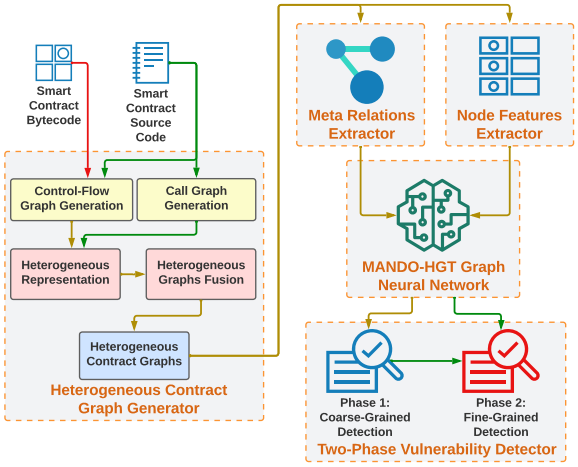

Fig. 3. MANDO-HGT Overview. The process flows indicated by yellow arrows are for both the bytecode and source code of the input contracts, while green arrows are for source code only.

vertex set V and an edge set E. $\tau : V \rightarrow A$ is a node-type mapping function and $\phi : E \rightarrow R$ is an edge-type mapping function. A and R denote the sets of node types and edge types, and $|A| \geq 2$ and $|R| \geq 1$.

Smart contract code, either source code or bytecode, is processed by the first component of Figure 3, **Heterogeneous Contract Graph Generator**, and translated into heterogeneous graphs based on control-flow graphs (CFGs) and/or call graphs (CGs). In MANDO-HGT, we use Slither [7] to analyze source code and EtherSolve [67] to analyze bytecode respectively, to construct basic CFGs and CGs. In contrast to previous studies [12], [68] that only consider *homogeneous* forms of control-flow graphs where types of nodes and edges are not utilized, we retain most of the structure and semantics of smart contract code through *heterogeneous* graphs that preserve various node and edge types. In particular, we convert basic CFGs and CGs into heterogeneous forms, called *heterogeneous control-flow graphs* and *heterogeneous call graphs*, and fuse them into *heterogeneous contract graphs (HCGs)*.

Heterogeneous Control-Flow Graphs (HCFGs). For input bytecode, a CFG may involve all possible opcodes defined in the Ethereum yellow paper [69], but using all the opcodes as node types can induce much learning overhead. Based on EtherSolve [67], we define six primary types of nodes representing important opcode blocks, including $ROOT$, $BASIC$, $DISPATCHER$, $FALLBACK$, $LEAF$, and $EXIT$ in MANDO-HGT: $ROOT$ and $EXIT$ represent entry and end blocks, respectively; $DISPATCHER$, $FALLBACK$, and $LEAF$ are $BASIC$ blocks with some unique characteristics. Specifically, a $BASIC$ block is a sequence of opcodes executed sequentially between a jump destination (`JUMPDEST` opcode) and a jump instruction (`JUMP` or `JUMPI` opcode). A $DISPATCHER$ block is a $BASIC$ block with the last opcode being the return or stop opcode. A $FALLBACK$ block is a $DISPATCHER$ block that has no call data, and none of the hashes matches when executed (`REVERT` opcode). A $LEAF$ block has the last opcodes being `REVERT`, `SELFDESTRUCT`, `RETURN`, `INVALID`, and `STOP` and has no successors, which means jumping to the END block in the CFG. In addition, three edge types are used to describe sequential ($NEXT$) or branching ($TRUE$ and $FALSE$) connections between nodes. Notably, the `JUMPI` opcode plays a vital role in a conditional branching structure; we add the edge type $FALSE$ for a branch that leads to the following opcode block when the branch condition is false, and the $TRUE$ edge type is for the true branch, which is the argument of the `PUSH` opcode interpreted as the destination offset for the `JUMPI`. Thus, a smart contract can be converted to a heterogeneous control-flow graph (HCFG). Figure 2 shows the generated HCFG for the runtime bytecode of a buggy contract.

For input source code, a CFG may involve many types of statements or lines of code. We use typical statement types as the node types for source code's HCFGs, such as $ENTRY\_POINT$, $EXPRESSION$, $NEW\_VARIABLE$, $RETURN$, IF, END_IF, IF_LOOP, and END_LOOP. Similar to bytecode, three edge types are used to indicate statements' sequential or branching nature, such as $NEXT$, $TRUE$, and $FALSE$. Figure 1 (Part C) shows a sample HCFG generated for the `Collect` function in the `MY_BANK` contract.

Due to the capabilities and limitations of the tools for generating CFGs (Slither [7] for Solidity source code and EtherSolve [67] for EVM bytecode), a bytecode's HCFG represents the control flows throughout an entire smart contract, while a source code's HCFG is only for one function of a contract. To integrate the HCFGs for all functions of a contract, we also utilize call graphs for source code as explained below.

Heterogeneous Call Graphs. A call graph (CG) represents the invocation relations among functions in one or multiple smart contracts. There are two basic forms of calls in smart contracts that the MANDO-HGT framework considers: *internal calls* for function calls within the same contract and *external calls* for function calls across contracts, represented by two edge types $INTERNAL\_CALL$ and $EXTERNAL\_CALL$ respectively. In addition to the typical function node type $FUNCTION\_NAME$, we also employ the $FALLBACK\_NODE$ node type to represent fallback functions that are executed if a function identifier to be called does not match any accessible function in a smart contract or if insufficient data was provided for the function call. Such fallback functions are directly or indirectly related to numerous Ethereum smart contract vulnerabilities [70]. Figure 1 (Part B) shows such a heterogeneous call graph.

We also use Slither to process each smart contract source code to produce its heterogeneous call graph and add the explicit types to the nodes and edges.

Heterogeneous Contract Graphs (HCGs): Fusion of Heterogeneous Call Graphs and Heterogeneous Control-Flow Graphs. The topologies of these two kinds of graphs for a smart contract source code can be merged into a global graph, to facilitate the graph learning process later. In MANDO-HGT, the sub-component **Heterogeneous Graphs Fusion** is for this purpose: for each node in the heterogeneous call

graph that represents a function, a bridging edge is added to link this node to the entry node of the heterogeneous control-flow graph for the function (as illustrated by the double-arrow edge between Figure 1 Part B and Part C). We call such a fused graph a *heterogeneous contract graph (HCG)*. For bytecode, since the heterogeneous control-flow graph generated by EtherSolve has represented the entire smart contract, our **Heterogeneous Graphs Fusion** sub-component directly utilizes the bytecode's HCFG as the fused HCG.

B. Meta Relations Extractor

Definition IV-B.1 (Meta Relation)**.** A meta relation of an edge $e = (s, t)$ from a source node s to a target node t is indicated as $\langle \tau(s), \phi(e), \tau(t) \rangle$, with $\tau(s)$ and $\tau(t)$ representing the node type of *s* and *t*, respectively, and $\phi(e)$ representing for the edge type of e. A metapath can refer to a sequence of such meta relations for a sequence of connected edges.

The component **Meta Relations Extractor** of MANDO-HGT extracts customized meta relations from the generated *heterogeneous contract graphs (HCGs)*. The main advantage of extracting meta relations is avoiding the explosion of all possible node and edge types combinations in the traditional approaches that use metapath [19], [20] as the number of node types and edge types in the graphs is dynamic and can be up to eighteen node types and five edge types.

We also add meta relations through reflective connections between adjacent nodes, e.g., the relation between two adjacent nodes of the types EXPRESSION and END_IF in Figure 1 can be described by both $\langle EXPRESSION, next, END\_IF \rangle$ and $\langle EXPRESSION, back, END\_IF \rangle$. HCGs are predominantly tree-like, with only a few back-edges created by LOOP-related statements. Adding the reflective relations increases the comprehensiveness of the extracted meta-relations, and improves the stable operability of the heterogeneous graph transformer (HGT) used in MANDO-HGT because the original architecture of each HGT layer requires at least two source nodes for one target node [21] and, without reflective relations, many nodes having only one source node (e.g., the two EXPRESSION nodes in Figure 1) would be ignored during training.

C. Node Features Extractor

The main goal of this extractor is to generate basic node features via one of the two following ways. (1) Generate node embeddings via some basic graph neural network without considering node and edge types. Omitting node and edge types is often reasonable for graph classification, as the connectivity topology among nodes is often sufficient to differentiate the graphs. We employ either homogeneous (e.g., node2vec [71]) or heterogeneous (e.g., metapath2vec [19]) graph neural networks in our evaluation (Section V). (2) Generate one-hot vectors based on the node types as the node features. These node features were used as initial embeddings for the MANDO-HGT's first layer to leverage the rich information independently *without* relying on any other neural network.

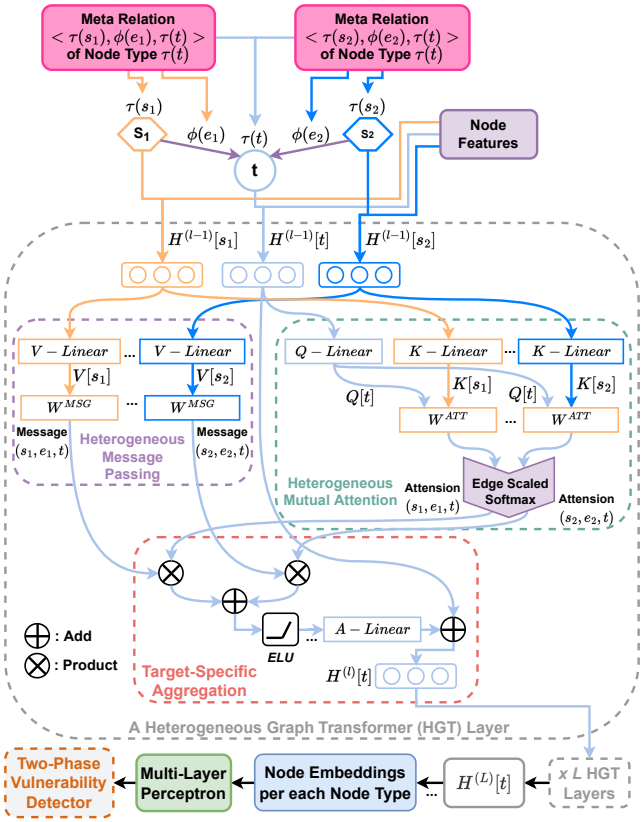

Fig. 4. The architecture of the MANDO-HGT Graph Neural Network.

D. MANDO-HGT Graph Neural Network

Figure 4 illustrates the architecture of the MANDO-HGT Graph Neural Network, based on Heterogeneous Graph Transformer (HGT) [21]. In MANDO-HGT GNN, we feed all pairs of meta relations of every target node, including their node types and node features, as inputs to one HGT layer. This is the major difference in our framework from the original HGT GNN. Such a mechanism allows MANDO-HGT to learn the inter-relation among our customized meta relations in HCGs. It is important since it disentangles complex node/edge relations for learning. The outputs of **MANDO-HGT GNN** are fed into the final components **Two-Phase Vulnerability Detector** to identify whether the smart contracts contain bugs and to find the bug locations in the contract source code.

Heterogeneous Graph Transformer (HGT) layer. The goal of this layer is to learn the *attention* of every pair of meta relations between a target node t and its neighbor source nodes s_1 and s_2 [21]. To achieve the goal, the architecture of Transformer [72] is employed with the target node t as the "Query" vector and its neighbors s_1 and s_2 as "Key" vectors. The attention is the output of the softmax layer applied to the concatenation of the output of h attention head [73]. Each attention head explores a different relation aspect of the two pairs of t with s_1 and t with s_2 by letting the embedding vectors of t with s_1 and t with s_2 go through the l-th GNN layer denoted by $H^{(l-1)}[t]$, $H^{(l-1)}[s_1]$ and $H^{(l-1)}[s_2]$ in

Figure 4. Specifically, there are three sub-components inside the HGT layer: *(1) Heterogeneous Mutual Attention:* The sub-component uses the Q and K linear transformations of target node t, and the two source neighbors s_1 and s_2 of t as the inputs, and the output is the attention or correlation probability of the two node pairs, i.e., s_1 or s_2 with t as well as the edge types $\phi(e_1)$ and $\phi(e_2)$ associated with the two given source nodes. The matrix W^{ATT} encodes multiple semantic relations of the pairs with the same node type. *(2) Heterogeneous Message Passing:* The input in this sub-component is the V linear transformations of the pair of source nodes s_1 and s_2, and the output is the multi-head message containing the distribution differences of nodes and edges with different types. We use matrix W^{MSG} to capture the edge dependency of each head. Note that the sub-component does not depend on the one above, so it can be processed simultaneously as the previous one. *(3) Target-Specific Heterogeneous Message Aggregation:* The sub-component is a multi-layer perception whose input is the aggregation of the outputs of the two components above, and the output is the contextualized representative vector $H^{(l)}$ of node t. Also, this sub-component uses an Exponential Linear Unit (ELU) as an activation function.

The target node t goes through L HGT layers to create the embedding vector $H^{(L)}[t]$. Such a mechanism ensures the final embedding vector of t is considered on multiple aspects through transformer architecture.

Optimization for Detection. For graph or node classification tasks, we use a multi-layer perceptron (MLP) with the softmax function as an activation function. The input of the layer depends on the prediction tasks. To reduce the effects of derivative saturation, we use cross entropy as the loss function during training, and the parameters of our model are learned through back-propagation with gradient descent algorithms.

E. Two-Phase Vulnerability Detector

This component comprises two primary phases: *Coarse-Grained Detection* and *Fine-Grained Detection*. While the former phase assesses clean versus vulnerable smart contracts at the contract level based on the input bytecode or source code, the latter phase determines the line locations of vulnerabilities in the contract source code. One of our new contributions is detecting vulnerabilities at the line level, whereas earlier learning-based methods [12], [14] only report vulnerabilities at the contract or function level.

1) Phase 1: Coarse-Grained Detection: This phase determines whether a smart contract is vulnerable. We employ the *heterogeneous contract graphs* and their embeddings for each input smart contract, and we train the MLP (Section IV-D) to predict whether a HCG is clean or vulnerable. The MLP can produce a confidence score for each input graph with respect to each bug type; the contract is classified/predicted as buggy when the confidence score for the graph with respect to a bug type is greater than 0.5. This classification helps reduce the search space by filtering out likely clean smart contracts prior to the second phase of line-level vulnerability detection.

Bug Types	# Total / Buggy Contracts	# Total Nodes (source / byte code)	# Total Edges (source / byte code)	# Buggy Nodes (source code only)
Access Control	114 / 57	13014 / 44475	10721 / 61896	7500
Arithmetic	120 / 60	17372 / 47967	14271 / 66020	10110
Denial of Service	92 / 46	13968 / 41066	11997 / 56711	8280
Front Running	88 / 44	22824 /51297	19761 / 71652	10008
Reentrancy	142 / 71	18898 / 46856	17614 / 64798	11238
Time Manipulation	100 / 50	16765 / 43424	15550 / 60464	10051
Unchecked Low Level Calls	190 / 95	17756 / 55103	14858 / 75950	7583
Total	846 / 423	120597 / 330188	119630 / 457491	64770

TABLE I
STATISTICS OF THE MIXED DATASET.

2) Phase 2: Fine-Grained Detection: For the smart contracts identified in the previous phase, the node embeddings of their *heterogeneous contract graphs* will go through node classification to determine if the nodes may be buggy. Similar to Phase 1, the MLP of the node classification step can produce a confidence score for each node in the input graph with respect to each bug type; a node is classified/predicted as buggy when the confidence score for the node with respect to a bug type is greater than 0.5. Note that the nodes correspond to statements or lines in source code, so we can identify the locations of vulnerabilities at the line level in source code [3].

V. EMPIRICAL EVALUATION

We publicize the datasets and our graph embedding models at https://github.com/MANDO-Project/ge-sc-transformer.

A. Dataset

Our evaluation is carried out on a mixture of three datasets: **(1) Smartbugs Curated** [23], [24] is a collection of vulnerable Ethereum smart contracts organized into nine types. It contains 143 annotated contracts having 208 tagged vulnerabilities. **(2) SolidiFI-Benchmark** [25] is a synthetic dataset of vulnerable smart contracts with 9369 injected vulnerabilities in 350 distinct contracts and 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*. Together with Smartbugs Curated, we have 423 buggy smart contracts in total since there are some contracts that could not be processed by Slither [7] or EtherSolve [67]. **(3) Clean Smart Contracts from Smartbugs Wild** [23], [24] is a set of 47,398 Ethereum smart contracts. We identified 2,742 contracts out of 47,398 that do not contain any bugs using 11 integrated detection tools of Smartbugs. Thus, we use the 2,742 contracts as a set of clean contracts.

The smart contracts in all the datasets are in source code form. We employ Slither [7] to traverse and generate the basic homogeneous forms of CFGs and CGs for the input source code. To get smart contract bytecode, we use Cryptic

[3]MANDO-HGT can also potentially detect bugs at the instruction level in bytecodes. However, when the bytecode instructions cannot be mapped back to source code lines, the detection results may not be readable by human developers. Thus we do not perform instruction/line-level bug detection for bytecode in this paper.

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Original Heterogeneous GNN	metapath2vec	48.43 72.80	54.89 69.52	64.13 69.46	66.79 69.86	64.28 62.38	60.73 64.88	62.74 69.23
Original Homogeneous GNNs	LINE	57.22 68.64	51.45 69.20	61.11 70.98	50.74 69.52	66.34 67.39	65.92 69.99	63.79 70.00
	node2vec	60.78 53.64	61.73 55.55	64.16 49.63	66.50 50.27	62.69 49.23	64.53 50.04	63.22 53.15
The best Buggy F1 scores of MANDO	Node features of the best scores	71.19 82.91	66.85 81.22	89.15 83.95	89.86 84.09	76.09 79.13	87.71 83.95	72.08 77.76
MANDO-HGT with Node Features Generated by	**NodeType One Hot Vectors**	82.86 79.46	88.13 **88.93**	89.33 85.93	92.69 **87.67**	93.84 **87.97**	95.68 81.39	80.11 76.14
	metapath2vec	**83.65** 78.56	88.86 86.93	88.91 86.69	93.70 86.15	**94.78** 87.05	95.89 81.89	**81.75** 77.07
	LINE	82.75 77.73	**89.41** 88.39	**89.89** 87.47	**95.23** 86.62	93.27 87.69	**95.99** 81.36	80.77 76.83
	node2vec	82.65 80.67	87.91 87.77	85.86 **87.63**	90.83 86.19	93.75 84.84	95.81 81.23	79.05 76.14

TABLE II

PERFORMANCE COMPARISON IN TERMS OF BUGGY-F1 SCORE ON DIFFERENT BUG DETECTION METHODS AT THE *contract* GRANULARITY LEVEL FOR BOTH *source code* AND *bytecode*. IN EACH CELL, THE FIRST NUMBER IS THE SOURCE CODE'S RESULT, AND THE SECOND WITH GREY SHADING IS THE BYTECODE'S RESULT. WE USE THE *Heterogeneous Contract Graphs* OF BOTH CLEAN AND BUGGY SMART CONTRACTS AS THE INPUTS FOR MANDO-HGT. THE BEST PERFORMANCE FOR EACH BUG TYPE IS IN BOLDFACE SEPARATELY FOR SOURCE CODE AND BYTECODE. FOR MANDO, WE ONLY REPORT THE BEST PERFORMANCE AMONG NODE FEATURE GENERATORS FROM THEIR PAPER [22], DUE TO SPACE LIMIT.

compiler [74], a Python wrapper of the Solidity compiler, with the Solc versions flexibly depending on the declared versions in the source code, to generate the runtime bytecode from these source files. Then we use the EtherSolve tool [75] to build basic CFGs for bytecode. We have also developed a component for transforming the traditional CFGs and CGs generated by EtherSolve or Slither to our heterogeneous CFGs and heterogeneous CGs and then fuse them into *heterogeneous contract graphs* before extracting meta relations and feeding them to the **MANDO-HGT GNN** component.

Also, we randomly take some smart contracts from the clean set and mix them with the buggy set. We keep a ratio of 1:1 between clean and buggy contracts, in order to have more balanced training/test datasets for both graph and node classification tasks, following practices used in other deep learning-based bug detection studies in the literature that use more or less balanced datasets, e.g., SySeVR [51], Russell *et al.* [45]. Table I shows the actual numbers of buggy contracts and total numbers of contracts used in our experiments for each bug type, as well as the total numbers of nodes and edges in the constructed heterogeneous contract graphs for source code and bytecode. Note that for the fine-grained line-level bug detection task, our approach requires line-level labels for the bugs but some other datasets, such as the ones of Zhuang *et al.* [12], Liu *et al.* [14], and eThor [37] are not suitable for our experiments because they only have coarse-grained contract- or function-level labels for the bugs.

Note that the employed Slither and Ethersolve parsers do not automatically generate clean or vulnerable labels for a node. Instead, we wrote automated scripts to label the nodes based on the lines of vulnerable code identified either manually by Smartbugs authors or injected by the SolidiFI tool. For instance, Smartbugs authors label Line 11 in Figure 1 as containing a *reentrancy* bug; then our scripts labeled the nodes with red text in the heterogeneous CFG and CG as vulnerable.

B. Evaluation Metrics

Our prediction results are binary (clean versus vulnerable) classification of a node or graph, so we measure the prediction performance using the commonly used F1 scores. An F1-score evaluates the performance of a model's prediction by taking the harmonic mean of precision and recall of the model for a given class label. As detecting bugs is the main interest of our evaluation, we measure the F1-score metrics to evaluate the performance of the models when classifying vulnerabilities; for the bug label, and we refer to this metric as *Buggy-F1* [4].

C. Baselines and Parameter Settings

Baselines. To show the advantages of heterogeneous graph learning over homogeneous graph learning, we use both of them in our evaluation: We apply *metapath2vec* [19] as the heterogeneous graph neural networks, while applying *node2vec* [71], *LINE* [76], and *GCN* [77] as homogeneous GNNs. Node embeddings generated from the baseline GNNs are used as the **Node Feature Extractor** for input node features of our **MANDO-HGT GNN** and the prediction baselines as well. Additionally, we use some variants of MANDO [22], a recent framework specialized in smart contract vulnerability detection based on GAT [73] and HAN [20] graph neural networks with multiple dynamic customized metapaths, as the baselines. We also compare our line-level source code bug detection method to six widely used smart contract vulnerability detection tools based on conventional software analysis techniques: *Manticore* [6], *Mythril* [78], *Oyente* [5], *Securify* [9], *Slither* [7], and *Smartcheck* [8].

[4]To measure the effects of imbalances between clean and buggy data, Macro-F1 is also often considered in the literature by averaging the F1 scores of all class labels. However, in our case, we used a ratio of 1:1 to balance the amount of clean and vulnerable contracts and found that Macro-F1 scores are very close to Buggy-F1 and omitted them in the paper.

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional Detection Tools	securify	13.0	0.0	18.0	53.0	23.0	24.0	11.0
	mythril	34.0	73.0	41.0	63.0	19.0	23.0	14.0
	slither	32.0	0.0	13.0	26.0	15.0	44.0	10.0
	manticore	30.0	30.0	12.0	7.0	9.0	24.0	4.0
	smartcheck	20.0	22.0	52.0	0.0	22.0	44.0	11.0
	oyente	21.0	71.0	48.0	0.0	20.0	24.0	8.0
Heterogeneous GNN	metapath2vec	35.46	68.70	60.64	80.65	71.66	67.51	26.06
Homogeneous GNNs	GCN	43.92	65.69	64.06	81.09	71.76	68.70	38.13
	LINE	53.59	68.61	62.28	83.06	74.78	70.76	7.10
	node2vec	44.94	67.84	63.92	81.84	71.52	67.81	34.26
The best buggy scores of MANDO	Node features of the best scores	81.98	84.35	82.12	90.51	86.40	90.29	84.81
MANDO-HGT with Node Features Generated by	**NodeType One Hot Vectors**	**89.46**	**91.18**	**86.81**	**94.02**	**92.59**	**95.04**	90.09
	metapath2vec	78.99	82.99	76.54	89.13	84.06	89.83	76.05
	GCN	87.76	88.86	83.73	92.96	89.59	92.91	89.47
	LINE	86.58	87.98	83.00	92.17	88.77	93.26	**90.89**
	node2vec	84.23	84.66	81.93	90.46	88.48	92.31	87.60

TABLE III
PERFORMANCE COMPARISON IN TERMS OF BUGGY-F1 SCORE ON DIFFERENT BUG DETECTION METHODS BASED ON *source code* AT THE *line* GRANULARITY LEVEL. THE BEST PERFORMANCE FOR EACH BUG TYPE IS IN BOLDFACE.

Parameter Settings. All models have their node or graph embedding size set to 128. We employ an adaptive learning rate ranging from 0.0005 to 0.01 for coarse-grained classification and from 0.0002 to 0.005 for fine-grained classification. For each target node and its each meta-relation pair that are fed to the **MANDO-HGT GNN**, we use two HGT layers [21] and set eight multi-heads whose hidden size is 128. We use their recommended settings for node2vec, LINE, GCN, metapath2vec, and MANDO to ensure the highest performance.

D. Experimental Results

In our initial experiments, we divided the 423 buggy contracts and 2,742 clean contracts into the training/validation/test sets using the 60%/20%/20% split ratio. However, some bug types in our dataset have fewer than 50 contracts, resulting in insufficient training/testing samples. In addition, we discovered that the loss value remains constant after a fixed number of epochs (100 and 50 epochs for Fine-Grained and Coarse-Grained tasks, respectively). In order to increase the training/test set sizes and maintain the proportion of vulnerable nodes in each set, we decided to split the contracts into only training/test sets using the 70%/30% ratio for all the bug types. For each setting, embedding method, and bug type, to ensure the robustness of our results we repeated the experiments 20 times with a different random seed and performed t-tests, and report the average results.

1) Coarse-Grained Contract-Level Vulnerability Detection: Table II shows the average results of 20 independent runs of the baselines and MANDO-HGT at the contract level for both source code and bytecode. We can observe that:

- MANDO-HGT and MANDO with node type generated by one hot vector [22] perform better than other methods, i.e., original heterogeneous and homogeneous GNNs for detecting bugs at source code or byte code levels. For instance, MANDO-HGT improves up to 34.52% compared to heterogeneous GNN for detecting *arithmetic* bugs with source code inputs. Moreover, no matter what methods are used to generate node feature, MANDO and MANDO-HGT frameworks still outperforms the baselines.
- Between MANDO-HGT and MANDO, the performance of MANDO-HGT is overall higher than that of the former for both source code and bytecode. However, the performance gaps between the two frameworks for some bug types are relatively small, e.g., 2.24% in access control bugs in the bytecode. For this reason, we apply a t-test to check if the performance of MANDO-HGT is statistically significantly better than MANDO based on the 20 runs of both models for each bug type. All of the t-values are positive while most p-values are less than 0.05, which implies that the performance of MANDO-HGT for most bug types is statistically significantly better than MANDO.
- It is clear that integrating node features via different graph neural networks inside MANDO-HGT outperforms all other GNN baselines, although it is unclear which node features perform the best. This observation is also applicable to various node features used by previous studies, e.g., MANDO. Hence, we believe that an architecture combining different GNNs is useful for classifying buggy contracts.

2) Fine-Grained Line-Level Vulnerability Detection: Table III shows the performances of MANDO-HGT and other baselines based on source code at the *line level*. From the results, we observe that:

- MANDO-HGT generally outperforms conventional analysis-based bug detection tools and basic GNNs. The performance improvements are around 10.96%–76.89% in Buggy-F1 scores, for different bug types. For example, for the *time manipulation* bugs, we got a 95.04% Buggy-F1 score, considerably higher than the best 70.76% among the baseline conventional tools and basic GNNs. Some conventional detection tools in Table III hardly function (Buggy-F1=0%) for certain vulnerability types due to their

inherent limitations in relying on predefined patterns that are incapable of capturing these vulnerabilities.

- In MANDO-HGT, node features generated by one hot vector are better than other node feature generating methods for most bug types. The only exception is the node features generated by LINE in MANDO-HGT for detecting *unchecked low level calls* bugs. For example, for the *reentrancy* bugs, we got the best Buggy-F1, 92.59%, with the node features based on NodeType One Hot vectors. The models with node features generated by other methods are all lower.

E. Case Studies: Interpreting Vulnerability Prediction Results

To shed light on the reasons why MANDO-HGT can produce successful or failed predictions, we aim to identify certain correlations between MANDO-HGT's prediction results and the actual semantics of smart contract code in this section. We use a post-hoc local model-agnostic interpretable framework, GraphSVX [27], a state-of-the-art method for graph interpretability, based on Shapley value [79] for graph neural networks to interpret the behaviors and confidence of our model's predictions versus the input smart contract source code. We also evaluated several other XAI methods, including the original SHAP, and found that the results of GraphSVX were better than others since GraphSVX considers more of the structures of graphs while the SHAP only considers individual object embeddings. Shapley values of an individual feature j is $\phi(val(j))$ providing a proxy to assess the overall significance of the feature j to an output of a data point by averaging the marginal contribution of the feature across all possible coalitions where the feature presents. In MANDO-HGT, when making a prediction on a *focal* node, we consider all neighbors N of this node as the features which could impact the prediction on the *focal* node. We obtain the Shapley value by:

$$\phi(val(j)) = \sum_{S \subseteq \{1,\dots,N\}\setminus\{j\}} \frac{|S|!\times(N-|S|-1)!}{N!}(val(S\cup\{j\}) - val(S)),$$

where $S \subseteq \{1,\dots,N\}\setminus\{j\}$ are the possible coalitions of node j's neighbors, and $val(S) = \mathbb{E}\left[f(\mathbf{X}) \mid \mathbf{X}_S = \mathbf{x}_s\right] - \mathbb{E}[f(\mathbf{X})]$ with $\mathbb{E}[f(\mathbf{X})]$ being average prediction of dataset $\mathbf{X}$.

We extract neighbor nodes of a focal node and compute their marginal contribution towards the prediction for the focal node, and use some random masking strategy [27] to select subsets of them to calculate their Shapley values more efficiently. The Shapley value of a node implies the contribution of the node to the focal node, and a node's confidence refers to the confidence score of the model's prediction for the node. A node is considered buggy if its confidence score is greater than 0.5 with respect to a bug type (cf. Section IV-E).

For any given node used as a focal node for examination, we calculate all of its neighbors' Shapley values to find out their effect on the model's bug prediction result. Larger Shapley values indicate the neighbor's higher impact on the focal node. In the following, we explain several cases where MANDO-HGT makes correct or incorrect predictions to further study possible correlations between the Shapley values of a focal node's neighbors and its meta relations. For instance, a buggy

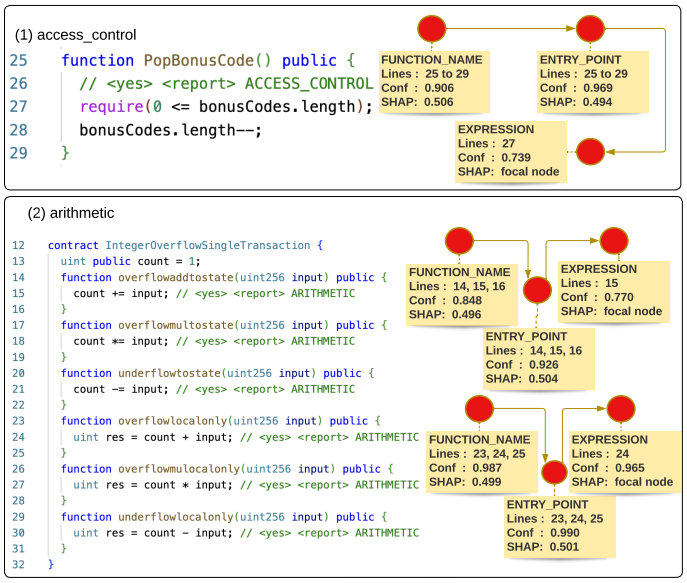

Fig. 5. True positive cases of *access control* and *arithmetic* samples.

node might belong to several meta relations, and if there is a meta relation that frequently contains nodes of a certain bug type and contributes the highest Shapley value to the focal node, the focal node's bug prediction confidence might be high. On the contrary, if some meta relations contain clean nodes contributing high Shapley values to the focal node, the bug prediction confidence for the focal node might be low.

From Figure 5 to Figure 8, subgraphs of the heterogeneous contract graphs are shown next to their source code. The nodes are filled with either red or green, indicating either buggy or clean predictions by our model for the nodes. The circle or triangle shapes of the nodes stand for buggy or clean nodes as indicated by the ground-truth labels. The yellow tag for each node contains information on the node type, its corresponding lines of source code, the confidence score of our model prediction (when the confidence is higher than 0.5, the node is predicted as buggy; otherwise, clean), and the Shapley value of that node to the focal node at the end of the arrow chain in each subgraph.

1) True positive cases: Figure 5-(1) shows a code snippet together with a sub-graph of its HCG that contains an *access control* bug at Line 27 corresponding to the focal node EXPRESSION in this case. We saw a meta-relation pair $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, EXPRESSION\rangle$ frequently appear in our samples. In this case, $FUNCTION\_NAME$ and $ENTRY\_POINT$ nodes correspond to the whole `PopBonusCode` function were predicted as a buggy node and had approximate Shapley values 0.506 and 0.494 contributing to the focal $EXPRESSION$ node, which implies that the focal node is likely buggy too. Indeed, MANDO-HGT correctly predicted the focal $EXPRESSION$ node as buggy.

Figure 5-(2) illustrates a smart contract that has arithmetic bugs at Lines 15, 18, 21, 24, 27, and

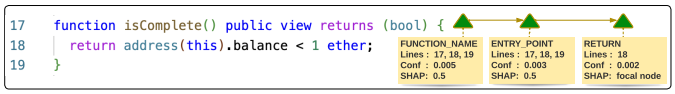

Fig. 6. True negative case of *access control* sample

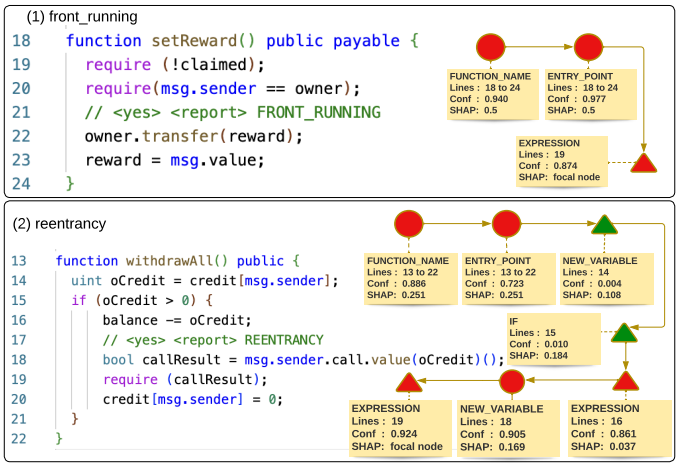

Fig. 7. False positive cases of *front running* and *reentrancy* samples

30 because the `input` variables in these functions are not checked for overflow or underflow before the operations are performed. One of the meta-relation pairs, $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, EXPRESSION\rangle$, appears three times in Lines 15, 18, and 21. Lines 24, 27, and 30 correspond to another meta-relation pair: $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, NEW\_VARIABLE\rangle$. The $FUNCTION\_NAME$ and $ENTRY\_POINT$ nodes represent an entire function and its entry point, indicating such bugs often happen at the beginning of a function with specific operations. MANDO-HGT is good at recognizing such frequently appearing bug patterns. Correspondingly, the Shapley values of these nodes to the focal buggy $EXPRESSION$ and $NEW\_VARIABLE$ nodes are around 0.5, which implies that relatively high Shapley values might reflect frequently occurring patterns for bug detection.

2) True negative cases: The code in Figure 6 generated a HCG with meta-relation pair $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, RETURN\rangle$. Its meta relations with $EXPRESSION$, $ENTRY\_POINT$, and $FUNCTION\_NAME$ node types also *differ* from those of *access control* bugs (e.g., the one in Figure 5-(1)). Besides, the Shapley values of $ENTRY\_POINT$ and $FUNCTION\_NAME$ nodes of function `isComplete` to the focal $RETURN$ statement, in this case, are 0.5, indicating that the two nodes (which are clean) have relatively significant impact on the prediction for $RETURN$. MANDO-HGT correctly predicted the focal node as a clean node (via a very low bug confidence score 0.002).

3) False positive cases: MANDO-HGT may wrongly predict some clean code as buggy. For example, for *front running* vulnerabilities, which are typically found in a statement that attempts to transfer a high amount for their transaction to be prioritized, they often involve an $EXPRESSION$ node, similar to the node for the line 22 in Figure 7-(1). However, some clean statements occasionally preceding the buggy line are also $EXPRESSION$ nodes, such as lines 19 and 20 in function `setReward`. The Shapley values from the buggy $FUNCTION\_NAME$ and $ENTRY\_POINT$ nodes are 0.5 and 0.5, which indicated that they have significant impact to that focal node, causing our model to mistakenly classify the focal node at Line 19 as buggy.

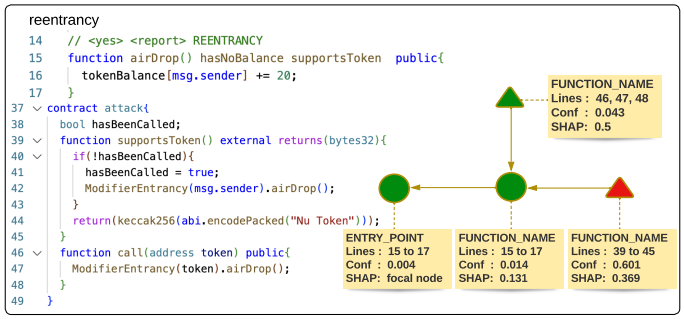

Fig. 8. False negative cases of *arithmetic* and *reentrancy* samples

Figure 7-(2) illustrates a similar false positive situation for the reentrancy bug type. The $NEW\_VARIABLE$ node corresponds to the code Line 18 defining a new variable and has a reentrancy bug in relation to the reduction of relevant `credit` to zero at Line 20. The focal $EXPRESSION$ node at Line 19 was wrongly predicted by MANDO-HGT, likely because it has data dependency with `callResult` defined at the buggy line 18. The buggy nodes like $FUNCTION\_NAME$, $ENTRY\_POINT$, and $NEW\_VARIABLE$ at Line 18 have relatively higher Shapley values of 0.251, 0.251, and 0.169 to the focal node at Line 19 than the Shapley values from the other two clean nodes (IF at Line 15 and $NEW\_VARIABLE$ at Line 14), 0.184 and 0.108. This indicates that the buggy nodes have more impact on the focal $EXPRESSION$ node at Line 19, causing MANDO-HGT to misclassify it as buggy.

4) False negative cases: MANDO-HGT may also miss certain vulnerabilities. For example, Figure 8 shows a *reentrancy* bug in the focal node $ENTRY\_POINT$ involving lines 15,16,17, but our model predicts it as a clean node. By considering the meta relations pair $\langle FUNCTION\_NAME, next, FUNCTION\_NAME\rangle$ and $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ of the focal node, we have the *clean* $FUNCTION\_NAME$ node containing lines 46, 47, 48 with 0.5 Shapley value having the greatest impact to the focal $ENTRY\_POINT$ node, higher than the 0.131 Shapley value from the buggy $FUNCTION\_NAME$ node containing lines 15,16,17, causing our model to predict the focal node as clean.

F. Limitations and Discussions

We know that the effectiveness of heterogeneous graph transformers relies on node/edge types and meta relations used,

besides various hyperparameters for training. The graphs used to represent syntactical and semantic information from smart contract source code or bytecode also significantly impact graph learning. Especially for bytecode, whose syntactical structure is flatter with more obscured semantic information than source code, suitable graph representations become more important for effective learning. Also, although our model can detect fine-grained bugs at the instruction level for bytecode as well, it would be better to map the detected bugs in bytecode back to source code lines for better readability when reporting the results to developers. We restrain our tool from reporting instruction-level bug detection results before we have a reliable way to make the results understandable in relation to their source code.

Lacking labeled data is always an issue when applying supervised learning to classification tasks, especially for smart contracts with limited sample buggy code for training. Although Smartbugs and SolidiFI datasets are useful, some smart contracts in the datasets were not annotated. Furthermore, the labels annotated in the datasets for some bugs may not always be objective and agreeable by all developers as there can be subjective and more than one interpretation of the root causes of a bug (e.g., bugs due to some missing lines of code); some labels are not fine-grained enough for each line or even each expression in code. Such inconsistent labels may hinder the training of our models. In our experiments, we excluded unlabelled data and manually checked some data samples and corrected a few inconsistent labels, and used balanced data sets for training and testing, to minimize the impact of inaccurate labels or imbalanced data.

In the experiments, some smart contracts could not be processed by Slither [7] or EtherSolve [67]. One of the main limitations of the tools is that they rely on the Solidity compiler to build the control-flow graph of the contracts, but there are issues related to the version compatibility of Solidity or the uses of optimized/obfuscated bytecode. For example, a smart contract written in an older version of Solidity is not supported by the compiler used by Slither or EtherSolve; the control-flow graph may not be built successfully.

VI. Conclusion & Future Work

This paper proposes a new learning-based vulnerability detection framework for Ethereum smart contract source code and bytecode, named MANDO-HGT, using heterogeneous graph transformer (HGT) techniques. In particular, it constructs heterogeneous contract graphs (HCGs) that represent control flow and function call relations of smart contracts, then defines customized meta relations based on node/edge types in HCGs, adapts HGT models to generate embeddings for nodes and graphs, and uses the embeddings to train classifiers to recognize various types of buggy code at the granularity levels corresponding to either individual contracts or individual lines of code. Our evaluation results on a curated smart contract dataset containing labeled vulnerabilities show that MANDO-HGT can significantly improve the accuracy of many previous vulnerability detection techniques, including best-performing learning-based and best-performing conventional analysis-based ones. The improvements in terms of the F1-score range from 0.74% to 76.89% for various bug types and detection techniques.

In the future, we will extend heterogeneous contract graphs to be more comprehensive graph representations of Ethereum smart contracts (e.g., data dependencies and contract calls) for both source code and bytecode and combine heterogeneous graph transformers with more graph learning techniques, especially those suitable for few-shot learning and handling inconsistent labels, for more accurate and usable vulnerability detection. It may also be possible and interesting to utilize results from conventional testing, analysis, and verification techniques to help better train and improve our approach.

References

[1] J. Frizzo-Barker, P. A. Chow-White, P. R. Adams, J. Mentanko, D. Ha, and S. Green, "Blockchain as a disruptive technology for business: A systematic review," *International Journal of Information Management*, vol. 51, p. 102029, Apr. 2020.

[2] B. Bhushan, P. Sinha, K. M. Sagayam, and J. Andrew, "Untangling blockchain technology: A survey on state of the art, security threats, privacy services, applications and future research directions," *Computers & Electrical Engineering*, vol. 90, p. 106897, 2021.

[3] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–44, 2021.

[4] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269.

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *the ACM SIGSAC conference on computer and communications security (CCS)*, 2016, pp. 254–269.

[6] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1186–1189.

[7] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019, pp. 8–15.

[8] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[9] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *25th ACM Conference on Computer and Communications Security*, 2018.

[10] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.

[11] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "KEVM: A complete formal semantics of the ethereum virtual machine," in *IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.

[12] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *IJCAI*, 2020, pp. 3283–3290.

[13] S. Han, B. Liang, J. Huang, and W. Shi, "DC-Hunter: Detecting dangerous smart contracts via bytecode matching," *Journal of Cyber Security*, May 2020.

[14] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[15] H. Zhao, P. Su, Y. Wei, K. Gai, and M. Qiu, "GAN-enabled code embedding for reentrant vulnerabilities detection," in *Knowledge Science, Engineering and Management*, 2021, pp. 585–597.

[16] Z. Bo, S. Chenhan, P. Xiaoyan, A. Yang, T. Juncheng, and Y. Anqi, "Semantic-aware graph neural network for smart contract bytecode vulnerability detection," *Advanced Engineering Sciences*, vol. 54, no. 2, pp. 49–55, 2022.

[17] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[18] Y. Sun and J. Han, "Mining heterogeneous information networks: a structural analysis approach," *Acm Sigkdd Explorations Newsletter*, vol. 14, no. 2, pp. 20–28, 2013.

[19] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 135–144.

[20] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The World Wide Web Conference*, 2019, pp. 2022–2032.

[21] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proceedings of The Web Conference*, 2020, pp. 2704–2710.

[22] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang, "MANDO: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities," in *9th IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2022.

[23] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 530–541.

[24] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: a framework to analyze solidity smart contracts," in *the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[25] A. Ghaleb and K. Pattabiraman, "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*, 2020.

[26] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[27] A. Duval and F. Malliaros, "Graphsvx: Shapley value explanations for graph neural networks," in *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 2021.

[28] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[29] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan, "GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," *IEEE Access*, vol. 8, pp. 99 552–99 564, 2020.

[30] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A model-based testing platform for smart contracts," in *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1601–1605.

[31] Y. Huang, B. Jiang, and W. K. Chan, "EOSFuzzer: Fuzzing eosio smart contracts for vulnerability detection," in *12th Asia-Pacific Symposium on Internetware*, 2020, pp. 99–109.

[32] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. Chan, "WANA: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection," in *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 926–937.

[33] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 747–766.

[34] S. So, S. Hong, and H. Oh, "$smartest$: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *30th USENIX Security Symposium*, 2021, pp. 1361–1378.

[35] ConsenSys, "MythX Tech: Behind the scenes of smartcontract security analysis," https://blog.mythx.io/features/mythx-tech-behind-the-scenes-of-smart-contract-analysis/, 2019. [Online]. Available: https://github.com/ConsenSys/mythril

[36] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1029–1040.

[37] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: Practical and provably sound static analysis of ethereum smart contracts," in *ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.

[38] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 51–78.

[39] ——, "EtherTrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.

[40] A. Wang, H. Wang, B. Jiang, and W. K. Chan, "Artemis: An improved smart contract verification tool for vulnerability detection," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2020, pp. 173–181.

[41] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.

[42] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *26th ACM ESEC/FSE*, 2018, pp. 912–915.

[43] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1695–1712.

[44] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[45] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *17th IEEE international conference on machine learning and applications (ICMLA)*, 2018, pp. 757–762.

[46] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 41–50.

[47] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[48] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[49] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[50] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *ICSE*, 2022.

[51] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[52] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[53] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[54] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *ICPC*, 2022.

[55] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[56] A. K. Gogineni, S. Swayamjyoti, D. Sahoo, K. K. Sahu, and R. Kishore, "Multi-class classification of vulnerabilities in smart contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing," *IOP SciNotes*, vol. 1, no. 3, p. 035002, 2020.

[57] T. H.-D. Huang, "Hunting the ethereum smart contract: Color-inspired inspection of potential attacks," *arXiv preprint arXiv:1807.01868*, 2018.

[58] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "ESCORT: Ethereum smart contracts vulnerability detection using deep neural network and transfer learning," *arXiv preprint arXiv:2103.12607*, 2021.

[59] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," *arXiv preprint arXiv:1811.06632*, 2018.

[60] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[61] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of CodeBERT," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 425–436.

[62] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[63] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[64] N. D. Bui, Y. Yu, and L. Jiang, "InferCode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[65] D. Vagavolu, K. C. Swarna, and S. Chimalakonda, "A mocktail of source code representations," in *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1296–1300.

[66] S. Yang, X. Gu, and B. Shen, "Self-supervised learning of smart contract representations," in *ICPC*, 2022.

[67] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 127–137.

[68] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," *arXiv preprint arXiv:2106.09282*, 2021.

[69] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[70] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[71] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 855–864.

[72] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[73] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[74] Crytic-compile, "Abstraction layer for smart contract build systems," https://github.com/crytic/crytic-compile, 2022.

[75] F. Contro, M. Crosara, and cmariano, "SeUniVr/EtherSolve: Version used for ICPC-2021 paper," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4607305

[76] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *WWW*, 2015.

[77] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[78] B. Mueller, "Smashing smart contracts for fun and real profit," in *9th annual HITB Security Conference*, pp. 2–51.

[79] L. S. Shapley, *17. A Value for n-Person Games*. Princeton: Princeton University Press, 2016, pp. 307–318. [Online]. Available: https://doi.org/10.1515/9781400881970-018