MANDO-LLM: Heterogeneous Graph Transformers with Large Language Models for Smart Contract Vulnerability Detection

NHAT-MINH NGUYEN*, Singapore Management University, Singapore
HOANG H. NGUYEN*, L3S Research Center, Leibniz University Hannover, Germany and The University of Tennessee at Chattanooga, USA
LONG LE THANH, Hanoi University of Science and Technology, Vietnam
ZAHRA AHMADI, L3S Research Center, Leibniz University Hannover, Germany
THANH-NAM DOAN, Independent Researcher, USA
DAOYUAN WU, Lingnan University, Hong Kong SAR, China
LINGXIAO JIANG, Singapore Management University, Singapore

Detecting vulnerabilities in smart contracts is vital for the security and reliability of decentralized apps. To facilitate vulnerability detection, contract codes, including bug patterns, are represented as heterogeneous graphs with various nodes and edges, like control-flow and function-call graphs. However, existing graph learning techniques struggle with large, complex graphs. This paper presents MANDO-LLM, a novel framework that combines heterogeneous graph transformers (HGTs) with large language models (LLMs) for detecting vulnerabilities in smart contracts represented as heterogeneous contract graphs built upon control-flow and call graphs. MANDO-LLM uses LLMs to capture code features from control-flow and call data, customizes HGTs to learn embeddings with specific node-edge meta relations, and employs classifiers for vulnerability detection in Solidity code at both contract and line levels. Our evaluation shows that MANDO-LLM significantly outperforms existing methods on real-world large-scale imbalanced datasets, with F1-score improvements from 0.59% to 80.72% at the contract level. It is also one of the first effective methods for identifying line-level vulnerabilities, with performance boosts ranging from 3.09% to over 95% across different vulnerability types. MANDO-LLM's versatility allows easy retraining for various vulnerabilities without needing manually defined patterns.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; • **Computing methodologies** → **Neural networks**; • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: vulnerability detection, smart contracts, source code, heterogeneous graph learning, graph transformer, graph embedding, large language model, code embedding

*Both authors contributed equally to this research.

Authors' Contact Information: Nhat-Minh Nguyen, nmnguyen@smu.edu.sg, Singapore Management University, Singapore, Singapore; Hoang H. Nguyen, dr.hhn@hoanghnguyen.com, L3S Research Center, Leibniz University Hannover, Hannover, Lower Saxony, Germany and The University of Tennessee at Chattanooga, Chattanooga, TN, USA; Long Le Thanh, attentionocr@gmail.com, Hanoi University of Science and Technology, Ha Noi, Vietnam; Zahra Ahmadi, zahra.m.ahmadi@gmail.com, L3S Research Center, Leibniz University Hannover, Hannover, Lower Saxony, Germany; Thanh-Nam Doan, me@tndoan.com, Independent Researcher, Atlanta, Georgia, USA; Daoyuan Wu, wu@ln.edu.hk, Lingnan University, Hong Kong SAR, China; Lingxiao Jiang, Singapore Management University, Singapore, Singapore.

ACM 1557-7392/2025/9-ART
https://doi.org/10.1145/3765751

1 INTRODUCTION

Blockchain-based smart contracts have found applications across various domains, including finance, e-commerce, healthcare, logistics, and law [26]. Any bug or security vulnerability in a deployed smart contract can lead to severe consequences for both developers and users [8, 14]. As a result, there is a high demand for security assurance techniques, particularly those for smart contract vulnerabilities.

Various studies have investigated vulnerability detection in smart contracts using conventional software testing, analysis, and verification techniques [23, 37, 48, 62, 63, 73, 84, 89]. These methods often require specific oracles or descriptions of the expected or unexpected patterns or semantics of smart contract code for analysis. Unfortunately, specifying patterns can be labor-intensive and challenging to adapt tools to evolving contract languages and vulnerability types. Furthermore, the computational complexity of these techniques renders them costly when repeatedly running on a large set of smart contracts in search of new vulnerability types [1]. Consequently, a new class of vulnerability detection techniques has emerged, based on machine learning and deep learning approaches [10, 27, 36, 61, 107, 112]. These techniques aim to encode various syntactic and semantic code information through syntax trees, control-flow graphs, or program dependency graphs. They also train automated classifiers to differentiate vulnerable code from non-vulnerable ones. Learning-based techniques reduce the need for manually specified patterns or specifications, making it easier to adapt to new code types and vulnerabilities as long as training data is available. However, existing code-learning techniques often treat nodes and edges *homogeneously*, neglecting fine-grained differences in node and edge types and their precise locations in the code's trees and graphs. They also ignore the semantics of the corresponding code snippets, which are critical attributes of each node on the global graph. This results in inadequate learning and low detection accuracy especially for identifying security vulnerabilities at the fine-grained line level.

The previous work MANDO-HGT [68] partially mitigated the challenges associated with employing the heterogeneous graph transformer (HGT) [40] techniques to learn the embeddings of the heterogeneous graphs in smart contract vulnerability detection. However, this method does not take full advantage of the input source code's capabilities because it only exploits graph topologies and meta relations but ignores considering code snippets such as important properties of nodes. Besides, with the rapid improvements in the quality and quantity of large language models (LLMs) for code embedding generation, they show the significant availability of effectively representing code snippets [43, 53, 95]. Therefore, we integrate LLMs into our proposed MANDO-LLM method as a potential approach to enhance the overall performance of its predecessor, MANDO-HGT.

Due to the pre-trained on numerous corpora and tasks, LLMs contain a wide knowledge of programming languages, which can not only capture both the features of syntactic and semantic information but also understand multiple programming languages that are indicated by some multilingual code generation models such as CodeGeeX [108], CodeGen [71]-[70]. Moreover, LLMs can handle unstructured code formats found in online forums or code repositories by Zhong *et al.* [110] while ensuring code security by Wang *et al.* [93]. We utilize the tokenizer of some LLMs, which can effectively tokenize various programming languages, including Solidity source code.

Although built as an extension of our previous work MANDO-HGT [68], besides reusing HGT layers as a core component in our graph neural network (GNN), MANDO-LLM has made some vital improvements, with its main contributions being as follows:

- Our proposed MANDO-LLM framework features a syntax-based graph generator supported by tree-sitter [88]. This integration offers several advantages, including avoiding version conflicts and reducing

[1] In the context of cybersecurity, vulnerabilities denote particular types of bugs that, when exploited, can result in significant security threats. According to Decentralized Application Security Project (DASP) https://dasp.co/, all the bug types discussed in this work may be categorized as vulnerabilities; however, only a limited number of instances of these bug types are actually exploitable. In this paper, we do not distinguish between the two terms and treat them as synonymous.

the effort to set up entire projects due to the compiler's and environment's independence as its predecessors [66–68].

- MANDO-LLM's heterogeneous graph generator can easily adapt to other programming languages by providing a grammar set [39]. This helps to enhance the information contained within the nodes of our generated *heterogeneous contract graphs*.
- MANDO-LLM leverages the ability of LLMs to handle effectively unstructured code format to embed the code snippet inside each node of the generated graphs.
- Due to the vast knowledge of LLMs in both syntactic and semantic terms, the extended information on each node in the heterogeneous contract graphs, such as structural positions and code fragment representations based on LLM code embeddings remained during learning processes, MANDO-LLM outperforms all traditional baselines and state-of-the-art methods.

The rest of the paper is organized as follows. Section 2 describes the background and motivation for the study. Section 3 surveys related work. Section 4 presents our approach MANDO-LLM. Section 5 presents the settings and results of our empirical evaluation, and discusses their interpretations. Section 6 concludes with future work.

2 BACKGROUND AND MOTIVATION

Motivating Sample Source Code.

Figure 1 (Part A) depicts a snippet of a smart contract written in Solidity, featuring a *reentrance* vulnerability. Part B presents the call graph (CG) of the contract, and Part C illustrates a portion of the control-flow graph (CFG) for the `collect` function within the given contract. The root of the issue is in line 11, particularly within the statement `msg.sender.call`. This control flow allows the `collect` function to be repeatedly invoked before the deduction of `balances` at line 12. Consequently, this arrangement permits `msg.sender` to receive more values than what is specified by `_am`. To effectively identify and address this so-called *reentrance* vulnerability, it is essential to consider the control-flow and call relations among `msg.sender`, `balances`, and `_am`. The *reentrancy* vulnerability often arises from the incorrect use of *IF* statements when invoking the `msg.sender` function (see Section 5.6), which can inadvertently trigger the smart contract's *fallback* function and make it vulnerable. Besides, our heterogenous CFGs and CGs can be derived from the abstract syntax trees. Thus, the generated graphs do not depend on specific compilers because they only rely on the grammar and syntax of programming languages.

Motivating Combination of Heterogenous Control-Flow Graphs and Call Graphs. While CFGs represent the control flows of a smart contract, they still are not able to capture the information outside the individual code blocks, such as functions or classes. CFGs typically comprise isolated sub-CFGs of individual blocks. To overcome this constraint and enhance the understanding of the relationship among the code blocks inside a smart contract, we integrate CGs into CFGs when CFGs assume a pivotal role by establishing edges whenever one code block invokes other blocks or external libraries. This fusion of CFGs and CGs creates a composite graph that can effectively capture more complex code patterns that link up to other control flows outside the current block.

Motivating of Syntax-Based Graph Generator. In prior approaches, MANDO [67] and MANDO-HGT [68] employed Slither [23] static analyzer tool to generate CFGs and CGs. The approach required the compilation of smart contracts, which occasionally precipitate version conflicts and the absence of necessary libraries. Even though the tool helps generate the graphs consistently, it needs more human effort to address compilation-related issues. To address this limitation, we used tree-sitter [88] instead, a syntax-based graph generation tool to establish CFGs and CGs from ASTs.

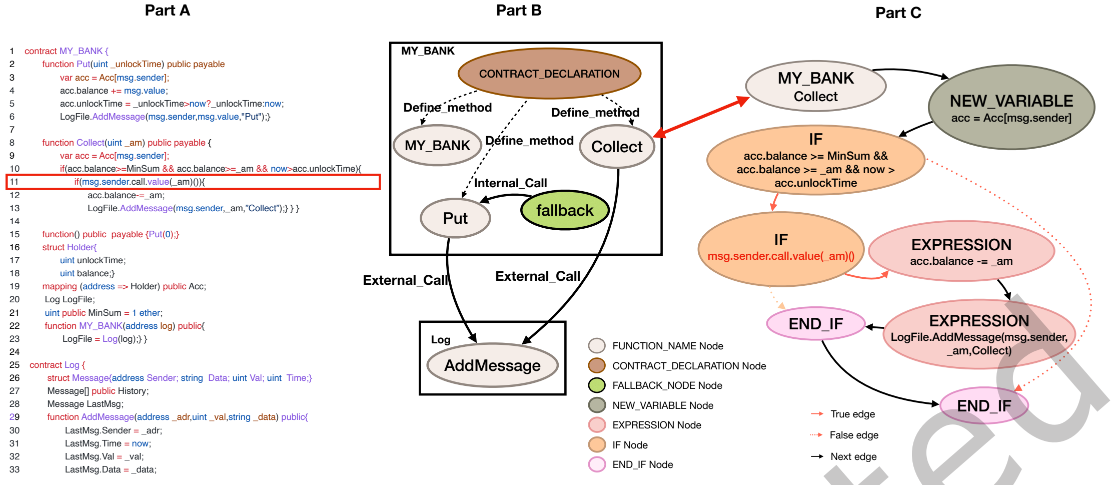
Motivating of Using Large Language Models' Tokenizers. LLMs is a type of language model that demonstrates remarkable capabilities in understanding programming languages across various tasks, including code generation from prompts and code explanation due to using massive amounts of data to learn millions or billions of trainable parameters. The application of LLMs for programming languages has naturally become a prevailing trend. Two main components play the most critical roles in an LLM: tokenizer and encoder. Tokenizer is the first module of LLMs and is responsible for splitting the input text into words, characters, subwords, or symbols as tokens. It then converts to token IDs using a look-up vocabulary table prior to feeding the array of token IDs into the billion weights of LLMs' encoder module to analyze the context, emphasize crucial parts, and understand the input's meaning based on a wide range of knowledge of pre-trained LLMs. As a result, tokenizers and encoders successfully embedded an input text into a much larger dimension vector that contains its own context and complex morphology and extends more features since the vocabulary was pre-trained on massive corpora including a wide range of text from the internet, such as articles, books, repositories which is necessary for the next downstream tasks such as code generation, code summarization, vulnerability detection, bug fixing. Through this training process, the tokenizers learn to recognize frequent words, characters, subwords, or symbols to create a comprehensive vocabulary for flawlessly chunking any given input to the sequences of token IDs. Consequently, when presented with a code snippet, LLMs can effectively analyze the code and discern patterns indicative of potential bugs within the provided programs. However, it is still unclear regarding the extent to which LLMs can assist us in bug identification and the specific types of bugs they can comprehend. In the scope of this research, we aim to address these questions, focusing specifically on the context of smart contracts.

Motivation for Combining HGT and LLMs. While Large Language Models (LLMs) such as CodeT5 and Starcoder demonstrate strong capabilities in understanding code semantics, they often lack awareness of a program's global structural context, especially for long-range dependencies and control/data flow across functions or contracts. On the other hand, traditional Graph Neural Networks (GNNs), including our earlier work MANDO-HGT, excel at modeling structural relationships but treat node content in a shallow way, often relying only on positional or type embeddings. MANDO-LLM bridges this gap through a synergistic combination: the LLM tokenizer enriches each node in the heterogeneous graph with fine-grained semantic embeddings derived from raw source code, while the Heterogeneous Graph Transformer (HGT) models higher-level relationships (e.g., control flow, internal/external function calls) via dynamically extracted meta-relations. This hybrid architecture allows our model to reason across both the content and structure of smart contracts, overcoming limitations of prior unimodal approaches.

3 RELATED WORK

3.1 Conventional Bug Detection Techniques

Detecting bugs and vulnerabilities has long been a crucial area of research across various computer science domains, such as programming languages, systems, cybersecurity, and software engineering. The emergence of increasingly diverse and complex languages and software systems necessitates the development of bug and vulnerability detection and prevention methods that are broad in application, scalable, and precise. Concurrently, as smart contracts in blockchain technology have received increasing popularity and reputation, the topic of vulnerability detection within these contracts is also gaining attention [5, 19]. Numerous studies have employed traditional program analysis and techniques from software engineering and security, such as testing/fuzzing [7, 42, 48, 59, 69], symbolic execution [18, 47, 63, 81, 96], and static/dynamic program analysis [23, 30, 31, 79, 89, 102], to detect specific bugs or vulnerabilities. For instance, OYENTE [62] employs symbolic execution to explore as many execution paths in smart contracts as possible, searching for four bug types. HONEYBADGER [87] focuses specifically on detecting honeypot smart contracts using symbolic execution and heuristic-based analysis, but it is limited in scope to identifying deceptive contracts that lure users into hidden traps, rather than detecting general-purpose vulnerabilities like access control or arithmetic bugs. Osiris [86] detects integer-related vulnerabilities in Ethereum smart contracts using abstract interpretation over control-flow graphs, but it is narrowly tailored to specific arithmetic issues. SmartCheck [84] uses static analysis to inspect smart contract code against established



```
1   contract MY_BANK {
2       function Put(uint _unlockTime) public payable
3           var acc = Acc[msg.sender];
4           acc.balance += msg.value;
5           acc.unlockTime = _unlockTime>now?_unlockTime:now;
6           LogFile.AddMessage(msg.sender,msg.value,"Put");}
7
8       function Collect(uint _am) public payable {
9           var acc = Acc[msg.sender];
10          if(acc.balance>=MinSum && acc.balance>=_am && now>acc.unlockTime){
11              if(msg.sender.call.value(_am)()){
12                  acc.balance-=_am;
13                  LogFile.AddMessage(msg.sender,_am,"Collect");} } }
14
15      function() public  payable {Put(0);}
16      struct Holder{
17          uint unlockTime;
18          uint balance;}
19      mapping (address => Holder) public Acc;
20       Log LogFile;
21       uint public MinSum = 1 ether;
22       function MY_BANK(address log) public{
23           LogFile = Log(log);} }
24
25   contract Log {
26        struct Message{address Sender; string  Data; uint Val; uint  Time;}
27        Message[] public History;
28        Message LastMsg;
29        function AddMessage(address _adr,uint _val,string _data) public{
30            LastMsg.Sender = _adr;
31            LastMsg.Time = now;
32            LastMsg.Val = _val;
33            LastMsg.Data = _data;
```

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.

rules about vulnerabilities and code issues. Sereum [75] is a runtime monitoring system designed to protect already-deployed contracts from *reentrancy* attacks; however, it is narrowly focused on this specific vulnerability class and does not generalize to other types of smart contract bugs. Several other research efforts [32, 37, 49, 72, 73, 85, 92] leverage formal verification to confirm the safety and functional accuracy of smart contracts according to certain human-defined specifications [28]. However, unlike our automatic bug pattern detection method, these security analysis techniques are designed to uncover specific vulnerabilities based on manually defined patterns or specifications. They often require a tailored implementation of testing, analysis, and verification algorithms specific to the smart contract language and type of vulnerability. This means their analysis algorithms can be significantly different, reducing their adaptability for new languages or vulnerability types. While our learning-based approaches also need specialized front-end code parsing and control-flow graph constructions, the graph-generating and graph-learning components in MANDO-LLM remain independent of the languages and can be applied to new types of vulnerabilities.

3.2 Learning-Based Bug Detection Techniques

Many software programs have explored learning from heterogeneous graphs for software analysis tasks like vulnerability detection and code search. Much of the prior research has typically either analyzed token sequences of code, data flow graphs, and control flow graphs for individual functions separately or treated cross-function relations independently from within-function relations, often due to scalability and design limitations in their methodologies. For instance, VulDeePecker [58] employs syntax structures and dependency slices to represent programs, utilizing prevalent neural network models to embed the programs and pinpoint vulnerability patterns for C/C++ programs. VulDeeLocator [56] builds on this by integrating attention-based granularity refinement to pinpoint line-level vulnerability locations more precisely. BGNN4VD [11] also leverages combined code representations in abstract syntax trees and control- and data-flow graphs to learn vulnerability patterns through bilateral graph neural networks for C/C++ programs. Several other studies have explored varying code representations and learning methods for source code in different languages [12, 15, 16, 54, 57, 76, 101, 111], with a scant few employing heterogeneous graphs for source code representation [104]. However, most existing approaches have largely been designed for languages other than Solidity and are unsuitable for Solidity smart contracts.

The DLVA [4] is a vulnerability detection tool for Ethereum smart contracts based on deep learning techniques to analyze bytecode input presented as sequential data. Notably, when the tool was trained on smart contracts annotated by Slither [23] static analyzer, DLVA exhibited ineffective performance compared to our manually labeled dataset.

MANDO-LLM and its predecessors, MANDO [67] and MANDO-HGT [68], are pioneering in their consideration of heterogeneous graph learning that combines control-flow graphs and call graphs, utilizing the learned embedding to identify vulnerabilities in Solidity smart contracts. Different from all previous studies, our frameworks, grounded in heterogeneous contract graphs, afford a more comprehensive representation of a smart contract's syntactic and semantic information, combining fine-grained individual statements and code lines within each function with cross-function call relations, facilitating pattern learning and search. Furthermore, they showcase scalability and adaptability to various vulnerability types.

3.3 Graph Embedding Neural Network Techniques

Some studies have utilized graph neural networks (GNNs) to detect vulnerabilities in smart contracts. Zhuang *et al.*[112] transformed the syntactic and semantic structures of each function in smart contracts into a contract graph, introducing a degree-free graph convolutional neural network besides leveraging expert patterns to comprehend the normalized graphs and detect vulnerabilities. Additional interpretability is offered by extracting vulnerability-specific expert patterns for graph encoding [60]. Wu *et al.*, with their tool, Peculiar [100], introduced a pretraining technique centered on customized data flow graphs of smart contract functions, aiming to identify reentrance vulnerabilities. Their methods, however, encounter several limitations: the dependency on expert patterns and a graph generator that only handles certain predefined Major and Secondary functions before graph creation results in less efficient graph generation process compared to our approaches in our MANDO-LLM and its forerunners, MANDO [67] and MANDO-HGT [68]. Furthermore, using predefined patterns restricts them to detecting only two specific bugs, Reentrancy and Time Manipulation, in Solidity source code. In contrast, our frameworks employ a heterogeneous graph structure, enabling a more general and flexible exploration of various vulnerability types without needing predefined patterns.

Other studies leverage different forms of embeddings, whether in code snippets or generated graph/tree structures. For instance, SmartEmbed [27] uses serialized structured syntax trees, training word2vec, and fastText models to detect vulnerabilities. SmartConDetect [46] views code fragments as distinct token sequences and employs a pre-trained BERT model to identify patterns that might be vulnerable. Additionally, Zhao *et al.* [107] apply word embedding, similarity detection, and Generative Adversarial Networks (GAN) to dynamically reentrance vulnerabilities.

Several heterogeneous graph models have been proposed in prior work. R-GCN [78] introduces relation-specific transformations for message passing over multi-relational graphs, and HGAT [103] applies node-level and type-level attention mechanisms for modeling heterogeneous information networks. However, both rely on a fixed schema of node and edge types. In contrast, MANDO-LLM dynamically generates meta-relations based on the actual node and edge types present in each smart contract graph. This flexibility allows our model to naturally adapt to unseen or evolving structures without requiring manual definition of relation types, making it more scalable and expressive in real-world settings.

While existing deep and graph neural network techniques have mitigated issues by learning bug patterns from specific code representations, such as syntax trees and data/control dependency graphs, they have typically treated trees/graphs as flattened sequences or disjointing conventional graphs. They have not leveraged particular kinds of control flow and call relations in the contract code to grasp their semantics more comprehensively. Also, they often regard nodes and edges in tree- and graph-representations of source code *homogeneously*, overlooking the fine-grained differences in their types and locations. As a result, they typically only search for vulnerabilities at the coarse-grained, whole-graph level, which does not offer the precision needed to identify the exact lines where vulnerabilities exist. Contrasting with the existing methods, our unique graph encodings in MANDO-LLM are able to accurately capture vulnerability patterns and locate fine-grained vulnerabilities at the line level.

3.4 Large Language Models

GraphCodeBERT [35], CodeBERT [24], UniXcoder [34], CodeReviewer [55] are the millions of parameters LLMs were trained on extensive corpora and datasets of programming languages that can be adapted to vulnerability detection task. However, it is important to note that these LLMs were not pre-trained on Solidity smart contracts, leading to decreasing performance when evaluated on our datasets. Furthermore, many LLMs within one billion to hundreds of billions of parameters are available on the Huggingface platform [98]. These LLMs were pre-trained on a larger number of programming languages and tasks such as code generation, Code summarization, and code completion. As a result of a wide range of pre-trained phases, these models can be adapted effectively to our challenge. CodeT5 [95] is an LLM pre-trained on CodeSearchNet [44] dataset which leverages the code semantics, seamlessly supports both code understanding and generation tasks and allows for multi-task learning. Starcoder [53] is an LLM designed specifically for 80+ programming languages from The Stack (v1.2) [51] dataset with opt-out requests excluded. GPTScan [82] is a recent vulnerability detection approach using GPT after breaking each logic vulnerability type into scenarios and properties to enhance accuracy. Although these existing LLMs are huge and pre-trained on many programming languages and tasks, they perform inadequately on vulnerability detection tasks and Solidity smart contracts because they are not specifically designed for Solidity vulnerability detection tasks.

4 APPROACH

We present the MANDO-LLM framework, which consists of five key components outlined in the grey boxes in Figure 2: the *Heterogeneous Contract Graph Generator*, the *Meta Relations Extractor*, the *Node Features Extractor*, the *MANDO-LLM Graph Neural Network*, and the *Two-Phase Vulnerability Detector*. The input to MANDO-LLM consists of the source code of one or more Ethereum smart contracts, and the output encompasses bug predictions for both contract-level and line-level issues, along with their associated bug types for the input contracts. In the following, we explain the five components in detail.

4.1 Heterogeneous Contract Graph Generator

The first component in Figure 2, **Heterogeneous Contract Graph Generator**, is responsible for processing the source code of a smart contract. This component aims to represent the source code using a heterogeneous graph based on Control-Flow Graphs (CFGs) and/or Call Graphs (CGs). Distinguishing itself from previous studies [60, 112], which primarily focus on *homogeneous* forms of control-flow graphs that do not utilize node and edge types, our approach retains much of the structure and semantics of smart contract code through the use of *heterogeneous* graphs that encompass various node and edge types. We define a heterogeneous graph as follows:

Definition 4.1 (Heterogeneous Graph). A heterogeneous graph is a directed graph $G = (V, E, \tau, \phi)$, consisting of a 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 $|R| \geq 1$.

The generation of CFGs is initiated based on an Abstract Syntax Tree (AST) produced by a parser generator tool [88] and an incremental parsing library [39]. From generated ASTs, we implemented a driver to collect a pre-defined set of node types and establish edges that follow the control flow of the source code to create Control Flow Graphs (CFGs). Similarly, this component can extract the nodes that invoked to each other via the

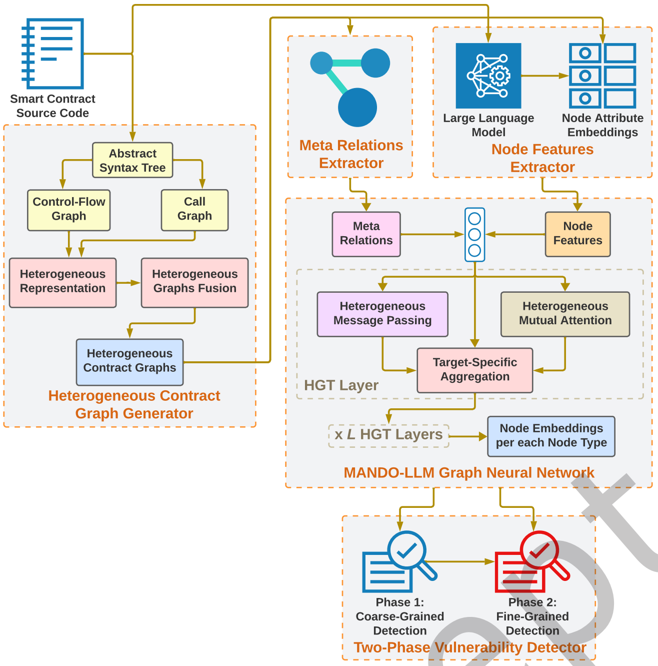

Smart Contract Source Code
Abstract Syntax Tree
Control-Flow Graph
Call Graph
Heterogeneous Representation
Heterogeneous Graphs Fusion
Heterogeneous Contract Graphs
Heterogeneous Contract Graph Generator
Meta Relations Extractor
Large Language Model
Node Attribute Embeddings
Node Features Extractor
Meta Relations
Node Features
Heterogeneous Message Passing
Heterogeneous Mutual Attention
Target-Specific Aggregation
HGT Layer
x L HGT Layers
Node Embeddings per each Node Type
MANDO-LLM Graph Neural Network
Phase 1: Coarse-Grained Detection
Phase 2: Fine-Grained Detection
Two-Phase Vulnerability Detector

Fig. 2. MANDO-LLM Architecture Overview.

function call edges to create Call Graphs (CGs), which are then adapted to CFGs to conduct a fusion form in the sub-component **Heterogeneous Graphs Fusion** in Figure 2 after translating to a heterogeneous representation. This fused graph is called a *heterogeneous contract graph (HCG)*.

Heterogeneous Control-Flow Graphs (HCFGs). A heterogeneous CFG may encompass various types of statements or lines of code. We employ typical statement types as node types within the source code's HCFGs, such as *EXPRESSION*, *NEW_VARIABLE*, *RETURN*, *IF*, *END_IF*, *IF_LOOP*, and *END_LOOP*. Four edge types are used to indicate the sequential or branching nature of statements, including *NEXT*, *TRUE*, and *FALSE*. An illustrative example of a generated HCFG for the `Collect` function within the `MY_BANK` contract is presented in Figure 1 (Part C). Notably, due to the capabilities and limitations of parser generator tools [88], an HCFG for a source code is generated on a per-function basis for a contract. To aggregate the HCFGs for all functions within a contract, we incorporate call graphs for source code, as further elaborated below.

Heterogeneous Call Graphs. A call graph (CG) represents the invocation relations among functions within one or multiple smart contracts. The MANDO-LLM framework acknowledges two primary forms of calls in the context of smart contracts: *internal calls*, which encompass function calls within the same contract, and *external calls*, which pertain to function calls that traverse across different contracts. These two call types are represented by two distinct edge types, namely *INTERNAL_CALL* and *EXTERNAL_CALL*. To distinguish between the functions of a smart contract and external functions, we introduce a node called *CONTRACT_DECLARATION* at the top of each contract. This node is linked to all its function definitions through *define method* edges. In addition to the typical function node type *FUNCTION_NAME*, we also incorporate the *FALLBACK_NODE* node type. This type is used to represent fallback functions that are executed in scenarios where a function identifier to be called does not match any accessible function within a smart contract or if insufficient data is provided for the function call. It is important to note that such fallback functions are directly or indirectly associated with numerous Ethereum smart contract vulnerabilities [13]. Figure 1 (Part B) illustrates an example of such a heterogeneous call graph for the *MY_BANK* contract.

Heterogeneous Contract Graphs (HCGs). The topological structure of these two types of graphs for a given smart contract can be integrated into a global graph to streamline the subsequent graph learning procedures. In MANDO-LLM, the sub-component **Heterogeneous Graphs Fusion** combines HCGs and HCFGs by establishing a connection between the *FUNCTION_NAME* nodes within both HCGs and HCFGs that represent the same function (as demonstrated by the double-arrow edge linking the *Collect* nodes in Figure 1 Part B and Part C).

4.2 Meta Relations Extractor

The *Meta Relations Extractor* component within MANDO-LLM is responsible for extracting customized meta relations from the generated *heterogeneous contract graphs (HCGs).* A meta relation is defined as:

Definition 4.2 (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 corresponding to a sequence of connected edges.

The main advantage of extracting these meta relations is to mitigate the potential explosion of all possible node and edge type combinations, a common issue in the traditional approaches employing metapath [20, 94]. This is particularly crucial since the number of node types and edge types in the graphs is dynamic and can encompass up to 18 node types and 5 edge types. Besides, the original architecture of the HGT layer also requires meta relations as the inputs to generate the embeddings [40].

Furthermore, we introduce meta-relations through reflective connections between adjacent nodes. For example, the relation between two adjacent nodes of 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 predominantly exhibit a tree-like structure, with only a few back-edges created by LOOP-related statements. The inclusion of reflective relations enhances the comprehensiveness of the extracted meta relations and improves the stable functionality of the heterogeneous graph transformer (HGT) employed in MANDO-LLM. This is because the original architecture of each HGT layer requires at least two source nodes for one target node [40]. Without reflective relations, many nodes with only one source node (e.g., the two *EXPRESSION* nodes in Figure 1) would be overlooked during training.

Clarification on Meta-Relation Definition and Impact. Unlike prior metapath-based approaches (e.g., MAGNET [97] and HeVulD [41]), which require manual predefinition of metapaths, MANDO-LLM employs a dynamic meta-relation extraction process. This strategy enables automatic and scalable relation modeling across diverse node and edge types, without the need to reconfigure or predefine relational patterns. Since our graph construction includes up to 18 node types and 5 edge types, the total number of possible meta-relations is up to 1,620. However, in practice, only a subset of these combinations appear in each contract graph. Empirically, we observe that the number of unique meta-relations per graph can range from dozens to several hundred, depending on the contract's structure. These relations are discovered on-the-fly during training and provided directly to each HGT layer. This dynamic handling significantly improves the model's adaptability and generalization to novel code structures, as confirmed in our experiments in Section 5.

4.3 Node Features Extractor

The primary objective of this extractor is to generate initial features for each node, enriching their properties within the graphs. In this module, we leverage the capabilities of large language models trained on vast corpora.

Although the whole encoder of the LLMs can produce an extremely rich information output, it requires costly resources. Additionally, fine-tuning an LLM to a new language for a new task also costs huge resources.

To overcome these constraints and prevent overwhelming the nodes with excessive information to conserve computational resources, we exclusively use the tokenizer part of the LLMs to encode and extract the node attribute features from the code snippets inside nodes rather than employing whole LLMs. In this way, we can leverage the advantages of LLMs' knowledge combined with source code understanding of GNN on the graph topologies of HCG, enabling MANDO-LLM to analyze smart contracts without high computational cost-effectively.

Each LLMs' tokenizer has its own vocabulary, which was pre-trained by a special technique. Byte-Pair Encoding [80] (BPE) is a compression technique known as the dark horse used by LLMs' tokenizers that works by iteratively merging the most frequent pairs of bytes which were split from massive text corpora into the token units such as words, characters, subwords, or symbols. The process would stop when the desired vocabulary size is reached or the tokenizer no longer finds frequent pairs of bytes to merge. As a result, the BPE process creates a set of variable-length tokens as a vocabulary well-suited for modeling code, as they can capture the fine-grained structure of code without sacrificing efficiency. Moreover, generating token units must be carefully balanced because a wide window of token units will capture richer context while dramatically increasing the vocabulary size or lack of understanding of the strange words. In contrast, short token units might be insufficient to maintain correct context while reducing vocabulary size, which can save computing costs. Due to well traded-off and pre-trained of the LLMs' tokenizer, we selected three LLMs as the baselines showed in Table 3 and 5 and their tokenizer based on three variants of BPEs as Node Feature Extractor that have certain benefits.

(1) Solidity-t5 [43]: The variant of the T5 [74] model within 770M trainable parameters is only one pre-trained on solidity smart contracts published on Huggingface platform. The model used SentencePiece BPE tokenizer within a wide vocabulary of 32,100 tokens. SentencePiece [52] was developed by Google AI for training efficient text understanding from unlabeled text and code. It was also trained on the Solidity dataset, making it well-suited for processing Solidity smart contracts.
(2) Starcoder [9]: A powerful LLM boasting 15.5 billion parameters pre-trained on 80+ programming languages, excluded solidity whose tokenizer is an original BPE [80] within even larger, 49,152 vocabulary size which has been pre-trained on more than 80 programming languages, making it versatile for various programming tasks.
(3) Codet5p-770m [77]: This model is a part of the CodeT5 [95] family and has been thoroughly evaluated across a wide spectrum of code understanding and generation tasks. Its tokenizer is code-specific BPE whose vocabulary size of 32,100 tokens from the massive dataset of code on GitHub allows it to capture the fine-grained structure of code without sacrificing efficiency.

We chose Solidity-t5 because it is specialized for the Solidity language. Meanwhile, the two models, Starcoder and Codet5p-770m, have been pre-trained on a wider range of programming language datasets and tasks, making them more versatile in a diverse range of code-related activities.

On the other hand, fine-tuning the LLMs in a new language for a new task requires many computing resources. To address the limitations, MANDO-LLM proposed a method to leverage a part of the LLMs, their tokenizer, to augment Graph Neural Network (GNN). This approach enables GNNs to inherit their pre-trained information while reducing the computing resources required for training from days to hours.

Due to time constraints, we avoid the overwhelming computation of tuning LLMs while efficiently capturing the rich semantic nuances of short code snippets inside nodes, we leverage smaller LLMs of a few billion parameters, instead of the recent powerful LLMs such as GPT [3], Gemini [2]. To compare with these more powerful LLMs, we use prompts instead, as shown in Section 5.4.

4.4 MANDO-LLM Graph Neural Network

Figure 2 illustrates the architecture of the MANDO-LLM Graph Neural Network, which is based on a Heterogeneous Graph Transformer (HGT) [40]. In our MANDO-LLM GNN, we input all pairs of meta relations for each target node, including their node types and node features, into a single HGT layer. This represents a significant departure from the original HGT GNN framework. This approach enables MANDO-LLM to learn the interrelations among our customized meta relations in HCGs. This is crucial for disentangling complex node/edge relationships, ultimately aiding in detecting actual bugs in a smart contract. The outputs from **MANDO-LLM GNN** are then passed on to the final component, the **Two-Phase Vulnerability Detector**, to identify whether the smart contract contains bugs and to locate these bugs within the contract source code.

Heterogeneous Graph Transformer (HGT) layer. The primary objective of this layer is to learn the *attention* among every pair of meta relations connecting a target node t and its neighboring source nodes s_1 and s_2 [40]. To achieve this, the architecture of Transformer [90] is employed with the target node t serving as the “Query” vector and its neighbors s_1 and s_2 as the “Key” vectors. The attention is computed as the softmax layer output applied to the concatenation of the outputs from h attention head [91]. Each attention head explores a different relation aspect of the two pairs of t and s_1 as well as t and s_2 by allowing the embedding vectors of t with s_1 and t with s_2 to pass through the l^{th} GNN layer, denoted as $H^{(l-1)}[t]$, $H^{(l-1)}[s_1]$ and $H^{(l-1)}[s_2]$. There are three sub-components within the HGT layer:

(1) *Heterogeneous Mutual Attention:* This 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 inputs. It produces the attention or correlation probability of the two node pairs, namely, 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:* In this sub-component, the input consists of the V linear transformations of the pair of source nodes s_1 and s_2, and the output is a multi-head message that contains distribution differences of nodes and edges with different types. We use the matrix W^{MSG} to capture the edge dependencies of each head. It is important to note that the sub-component is independent of the one above and can be processed simultaneously with the previous one.

(3) *Target-Specific Heterogeneous Message Aggregation:* This sub-component is a multi-layer perception (MLP) whose input aggregates the outputs from the two abovementioned components. It outputs the contextualized representative vector $H^{(l)}$ for the node t. Additionally, this sub-component uses an Exponential Linear Unit (ELU) as an activation function. The target node t undergoes L HGT layers to generate the embedding vector $H^{(L)}[t]$. Such a mechanism ensures that the final embedding vector for t considers multiple aspects through the Transformer architecture.

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

4.5 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 , the latter phase pinpoints the precise line locations of vulnerabilities in the contract source code. One of our notable contributions is the ability to detect vulnerabilities at the line level, a capability not present in earlier learning-based methods [61, 112], which only reported vulnerabilities at the contract or function level.

Phase 1: Coarse-Grained Detection. In this phase, the goal is to determine whether a smart contract is vulnerable. We employ the *heterogeneous contract graphs* and their corresponding embeddings for each input smart contract. We train an MLP (Section 4.4) to predict whether an HCG represents a clean or vulnerable contract. The MLP can produce a confidence score for each input graph with respect to each bug type. A contract is classified/predicted as buggy if the confidence score for the graph with regard to a particular bug type exceeds 0.5. This classification step aids in narrowing down the search space by filtering out likely clean smart contracts, preparing them for the second phase of line-level vulnerability detection.

Phase 2: Fine-Grained Detection. For the smart contracts identified as potentially vulnerable in the previous phase, we proceed to fine-grained detection. In this phase, the node embeddings of their *heterogeneous contract graphs* undergo node classification to determine if the individual nodes may be buggy. Similar to Phase 1, the MLP used in the node classification step generates a confidence score for each node in the input graph with respect to each bug type. A node is classified/predicted as buggy when its confidence score with respect to a specific bug type exceeds 0.5. Importantly, the nodes correspond to statements or lines in the source code, allowing us to identify the locations of vulnerabilities at the line level within the source code precisely.

5 EMPIRICAL EVALUATION

We publicize the datasets and our proposed models at https://github.com/MANDO-Project/ge-sc-llm.

Bug Types	# Total / Buggy Contracts	# Total of Nodes	# Total of Edges	# Buggy Nodes
Access Control	114 / 57	13432	15538	2977
Arithmetic	120 / 60	16157	17591	5395
Denial of Service	92 / 46	12909	14366	4106
Front Running	88 / 44	19415	22278	10400
Reentrancy	142 / 71	18312	19267	6570
Time Manipulation	100 / 50	15699	18564	5800
Unchecked Low Level Calls	190 / 95	17525	20952	2352
Total	846 / 423	113449	128556	37600

Table 1. **Dataset A** - Statistics of the Mixed dataset of the **Clean Contracts of Smartbugs Wild** with the **Smartbugs Curated** and the **SolidiFI-Benchmark** datasets .

5.1 Dataset

Our evaluation is carried out on a mixture of three datasets:

(1) **Smartbugs Curated** [21, 25] is a collection of vulnerable Ethereum smart contracts organized into nine types. It contains 143 annotated contracts with 208 tagged vulnerabilities.

(2) **SolidiFI-Benchmark** [29] 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*, *Reentrancy*, *Time Manipulation*, and *Unchecked Low-Level Calls*. Together with Smartbugs Curated, we have 423 buggy smart contracts.

(3) **DAppSCAN** [109] is a large-scale dataset with 39,904 Solidity files from 682 real-world DApp projects, featuring 1,618 SWC weaknesses [65]. For consistency in labeling, we convert the SWC weaknesses categories into seven types of vulnerabilities in datasets (1) and (2) based on the mapping by Shikah et al. [6]. After mapping, we have 614 buggy solidity files.

Bug Types	# Total / Buggy Contracts	# Total of Nodes	# Total of Edges	# Buggy Nodes
Access Control	244 / 122	22214	27019	5794
Arithmetic	372 / 186	51138	64166	10306
Denial of Service	316 / 158	38631	48891	7280
Front Running	252 / 126	33007	39629	14486
Reentrancy	326 / 163	36579	42519	10429
Time Manipulation	172 / 86	21431	25818	7067
Unchecked Low Level Calls	246 / 123	23940	29846	3821
Total	1928 / 964	226940	277888	59183

Table 2. **Dataset B** - Statistics of the Mixed datasets of the **Clean Contracts of Smartbugs Wild** with the **DAppSCAN** and the **SolidiFI-Benchmark** datasets .

(4) **Clean Smart Contracts from Smartbugs Wild** [21, 25] is a set of 47,398 Ethereum smart contracts. We identified 2,742 contracts out of 47,398 that do not contain any bugs based on eleven Smartbugs integrated detection tools. Thus, we use the 2,742 contracts as a set of clean contracts.

All smart contracts within the datasets are provided in source code form. We employ a parser generator tool called tree-sitter [88] and an incremental parsing library [39]. Using these two tools, we parse each source file into Abstract Syntax Trees (ASTs). Subsequently, we develop our driver, inspired by the Codeview Generator tool [1], to perform the following tasks: Generate Control Flow Graphs and Call Graphs for Solidity smart contracts as well as a fusion form combining these two graph types before extracting meta relations and feeding them to the **MANDO-LLM GNN** component.

Additionally, we introduce a random mixing strategy in which we select a subset of clean smart contracts from the **SmartBugs Wild** dataset and combine them with buggy contracts from **SmartBugs Curated**, **SolidiFI-Benchmark**, and **DAppSCAN**. This results in two composite datasets: **Dataset A**, which includes buggy contracts from both SolidiFI-Benchmark and SmartBugs Curated (see Table 1), and **Dataset B**, which includes buggy contracts from SolidiFI-Benchmark and DAppSCAN (see Table 2).

For the coarse-grained contract-level vulnerability detection task, we maintain a balanced ratio 1:1 between clean and buggy contracts, ensuring that our training and test datasets are more evenly balanced for graph classification tasks. This practice aligns with common practices in other deep learning-based bug detection studies in the literature, which also use reasonably balanced datasets, such as SySeVR [57] and Russell *et al.* [76]. In contrast, the fine-grained line-level vulnerability detection task focuses exclusively on the buggy contracts from **Dataset A** and **Dataset B**. Tables 1 and 2 summarize the number of buggy and total contracts, as well as the total nodes and edges in the constructed heterogeneous contract graphs across all bug types. Although the strategy of mixing with SolidiFI-Benchmark helps alleviate the class imbalance in both the original SmartBugs Curated and DAppSCAN datasets, Dataset B is still considered an imbalanced dataset, with buggy nodes accounting for only 26.07% of all nodes, while Dataset A exhibits a slightly more balanced distribution, with 33.14% of its nodes being buggy.

It is important to note that for the fine-grained line-level bug detection task, our approach requires line-level labels for the bugs; however, other datasets like those by Zhuang *et al.* [112], Liu *et al.* [61], and eThor [79] are not suitable for our experiments since they only provide coarse-grained contract- or function-level labels for the bugs. For instance, if Smartbugs authors label line 11 in Figure 1 as containing a *reentrancy* bug, our scripts would mark the corresponding nodes with red text in the heterogeneous CFG and CG as vulnerable.

5.2 Evaluation Metrics

Our binary prediction results distinguish between clean and vulnerable nodes or graphs. To assess the prediction performance, we measure our experimental results by two widely used metrics, the F1-score and Macro-F1 score. The F1 score evaluates a model's prediction performance by computing the harmonic mean of precision and recall for a specific class label by formula:

$$F1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

following by precision $= \frac{\text{TP}}{\text{TP+FP}}$ and recall $= \frac{\text{TP}}{\text{TP+FN}}$ where TP is true positive, FP is false positive, and FN is false negative predicted cases of the class.

On the other hand, the Macro-F1 score assesses the quality of predictions by averaging the F1 scores across all class labels. Since our primary focus is bug detection, we measure the F1-score to evaluate the model's performance in classifying vulnerabilities, specifically for the bug label. We refer to this specific metric as *Buggy-F1* [2].

In the literature, there are also other metrics for measuring the effectiveness of fault detection or localization [99, 106], such as EXAM (the percentage of statements to be manually examined until the first buggy statement is reached), top-N Accuracy (the percentage of true faults located within the first N reported faulty statements), MAR (Mean Average Rank), MFR (Mean First Rank), and RImp (Relative Improvement versus other fault localization methods). These metrics can be viewed as variants of precisions and recalls, and are intended to reflect the amount of effort human developers need to spend and investigate the code before confirming the faults. In this paper, we do not have actual human users to understand the reported fault code; simply relying on measurements with respect to the ground-truth fault locations without actual measurement of human efforts, different metrics may not show much difference [106]. Thus, this paper mainly uses the F1-scores.

5.3 Baselines and Parameter Settings

Baselines. To demonstrate the advantages of heterogeneous graph learning over some conventional Graph Neural Networks (GNNs), conventional Recurrent Neural Networks (RNNs), and Large Language Models (LLMs). We select three popular conventional GNNs models GAE [50], LINE [83], and node2vec [33], which use some basic graph learning techniques such as passing messages through the connections inside the graphs to learn the patterns and make the prediction about the graphs. We select the three classic conventional RNNs models Vanilla-RNN [22], LSTM [38], GRU [17], which receive the embedding of smart contract source code as a sequence of data and output the predicting probability of classes. The LLMs were not explicitly trained on solidity contracts for bug detection tasks, and we use the encoders of three LLMs: Solidity-t5, Starcoder, and Codet5p-770m to classify buggy code at both coarse-grained and fine-grained levels. We employ the pre-trained model's encoders to embed source code snippets, and then we fine-tune the last hidden state with classification layers. For the largest model, Starcoder, with 15.5 billion parameters, there were instances of long smart contracts that exceeded our server's capacity. In such cases, we divided the source file into individual smart contracts before encoding them. Finally, we calculated the average of the individual smart contracts to represent the source file. Additionally, we selected the best-performing result from MANDO-HGT [68], a recent framework specialized in smart contract vulnerability detection based on Heterogeneous Graph Transformer (HGT) [40] with multiple dynamic customized metapaths, as our methods of comparison. We also compared our line-level source code bug detection method to

[2] In the literature, Macro-F1 is also often considered to account for imbalances between clean and buggy data by averaging the F1 scores of all class labels. However, we have balanced the clean and vulnerable contract data at a 1:1 ratio in our case and found that Macro-F1 scores closely align with Buggy-F1.

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional GNNs	GCN	38.12	63.09	52.24	41.08	61.05	55.75	46.31
		40.50	57.70	58.92	45.11	55.24	51.29	47.62
	LINE	06.41	27.19	26.19	51.64	15.52	49.41	24.26
		36.16	41.64	32.86	35.30	33.79	37.46	40.60
	node2vec	53.82	61.96	54.95	75.46	36.30	60.66	56.07
		52.29	60.16	52.46	76.66	45.66	57.22	53.21
Conventional RNNs	Vanilla-RNN	23.53	53.49	56.52	04.76	57.73	57.14	44.66
		39.35	59.80	42.55	32.38	63.22	37.66	54.18
	LSTM	00.00	00.00	66.67	00.00	43.04	00.00	61.70
		31.34	32.89	33.33	31.73	56.42	30.43	00.00
	GRU	00.00	00.00	38.60	56.18	45.45	64.41	47.06
		32.35	31.54	50.07	41.30	55.58	32.20	56.44
LLMs Encoder	solidity-t5	85.00	82.55	88.33	79.29	83.96	85.68	86.06
		85.05	82.32	88.88	81.38	85.04	85.87	85.13
	starcoder	77.45	75.12	86.40	77.58	89.16	82.74	78.17
		77.78	79.27	86.78	81.46	89.37	84.50	80.61
	codet5p-770m	83.04	71.63	69.15	81.04	92.05	82.58	80.75
		83.27	76.67	68.49	82.77	92.20	82.93	81.99
The best Buggy F1 scores of MANDO-HGT	Node features of the best scores	86.36	83.59	88.61	93.28	92.44	92.37	82.97
		85.63	82.35	87.87	93.17	92.17	91.97	82.53
MANDO-LLM with Node Features Generated by LLMs' Tokenizer	solidity-t5	**87.13**	**88.02**	90.45	91.61	92.43	95.47	**86.65**
		86.12	**87.42**	**90.23**	90.94	92.23	94.91	**85.72**
	starcoder	87.60	84.97	88.61	93.29	**94.26**	**96.18**	82.33
		86.31	82.95	87.69	93.18	**94.30**	**95.97**	81.44
	codet5p-770m	86.71	86.82	**91.20**	**95.42**	91.67	95.14	82.23
		85.63	85.64	89.90	**95.54**	91.49	94.56	82.09

Table 3. Performance comparison in terms of Buggy-F1 score and Macro-F1 (in grey shading) score on different bug detection methods at the *contract* granularity level on the **Dataset A**. We use the *Heterogeneous Contract Graphs* of both clean and buggy smart contracts as the inputs for MANDO-LLM. The best performance for each bug type is in boldface. For MANDO-HGT, we only report the best performance among node feature generators from their paper [67].

six widely used smart contract vulnerability detection tools based on conventional software analysis techniques: *Manticore* [63], *Mythril* [64], *Oyente* [62], *Securify* [89], *Slither* [23], and *Smartcheck* [84].

LLM Prompts. We prompt the industrial LLMs: DeepSeek, GPT-4.0, and Gemini Pro for vulnerability detection at both coarse-grained contract-level on the **Dataset B** (shown in Table 4) and fine-grained line-level on the buggy contracts of the **Dataset B** (shown in Table 6). The prompts can be found in Appendix A.1.

Parameter Settings. All models in our experiments use an input node feature size of 512. We used double size compared with the reimplement experiment of HGT [40] by DGL [45] to keep more information of the code inside a node from LLM's tokenizer because LLM's tokenizer output a huge token dimension. At the same time, we reduced half of the hidden size from 256 to 128 to meet our computing resources. We noticed the decrease did not considerably affect our results. We incorporate 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 corresponding meta-relation pair, fed to the **MANDO-LLM GNN**, we apply two HGT layers and configure eight multi-heads, followed by the original paper's recommendations [40]. We used a cross-validation approach to run the experiment with 5-fold for the coarse-grained classification and 10-fold for the fine-grained classification. All experiments are conducted on a server with two GPU A100-PCIE-80GB units and CUDA Version 11.2. One of the advantages of our approach is that it does not require a powerful server since we only use the tokenizer of the LLMs instead of passing the input through the LLMs. In contrast, the other approaches that directly use LLMs for bug detection may require a powerful server and a special process for large smart contracts (see Section 5.3).

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional GNNs	GCN	00.00	38.20	65.24	56.60	62.93	00.00	00.00
		33.33	48.73	32.62	28.30	31.46	30.66	31.48
	LINE	00.00	66.26	00.00	64.07	74.35	00.00	62.96
		33.92	34.85	34.93	44.28	37.17	32.46	31.48
	node2vec	59.40	62.96	61.31	68.96	71.89	00.00	69.02
		36.08	64.24	30.65	34.48	35.94	29.72	34.51
Conventional RNNs	Vanilla-RNN	63.64	32.43	76.11	62.92	65.55	31.82	67.37
		67.00	46.16	78.48	66.86	68.44	49.95	68.64
	LSTM	00.00	68.66	64.17	40.00	00.00	00.00	64.38
		32.41	51.85	32.09	54.09	31.41	30.30	32.19
	GRU	66.67	43.64	00.00	67.35	51.49	00.00	00.00
		61.57	55.33	30.60	68.29	60.53	32.65	32.65
LLMs Encoder	solidity-t5	00.00	56.41	63.30	63.06	70.19	00.00	49.35
		35.08	28.20	31.65	31.53	35.09	35.49	47.61
	starcoder	05.12	71.67	68.05	70.08	67.56	00.00	63.63
		35.59	37.79	34.02	35.04	33.78	34.17	67.18
	codet5p-770m	00.00	60.00	62.31	67.82	65.75	00.00	42.69
		32.11	30.00	31.15	33.91	32.87	36.58	28.12
Prompting LLM	DeepSeek	54.50	44.40	52.60	44.40	58.80	00.00	00.00
		74.40	70.70	75.50	71.60	76.80	50.00	47.60
	GPT-4.0	54.50	47.00	15.00	25.00	66.70	00.00	00.00
		74.30	71.90	57.10	62.50	83.30	50.00	44.00
	Gemini Pro	30.80	23.50	00.00	15.40	50.00	00.00	74.10
		61.10	57.40	47.70	51.10	50.00	50.00	69.60
The best scores of MANDO-HGT	Node features of the best scores	83.63	80.99	87.88	88.46	88.52	87.97	82.35
		81.63	81.14	87.49	87.98	89.33	87.74	81.99
MANDO-LLM with Node Features Generated by LLMs' Tokenizer	solidity-t5	**92.30**	88.89	90.91	90.20	89.55	94.12	88.37
		91.80	89.18	90.47	90.00	89.22	94.28	**89.64**
	starcoder	86.79	**89.47**	90.91	**90.57**	88.89	94.12	83.33
		85.61	**89.33**	90.62	**90.18**	89.37	94.28	83.67
	codet5p-770m	85.71	84.93	**94.12**	88.00	**92.54**	**97.14**	**88.46**
		85.71	85.32	**93.73**	88.23	**92.30**	**97.06**	87.98

Table 4. Performance comparison in terms of Buggy-F1 score and Macro-F1 (in grey shading) scores on different bug detection methods at the *contract* granularity level on the **Dataset B**. We use the *Heterogeneous Contract Graphs* of both clean and buggy smart contracts as the inputs for MANDO-LLM. The best performance for each bug type is in boldface.

5.4 Experimental Results

We use cross-validation and an early stopping method to avoid overfitting. We set the patience parameter to 7 and the delta parameter to 0.001. Such a setting helps us stop our training process if the model's loss does not improve by more than 0.001 for seven consecutive times. The final results in the table represent the average scores of all folds (5 for coarse-grained and 10 for fine-grained).

5.4.1 Coarse-Grained Contract-Level Vulnerability Detection. Table 3 and 4 display the Buggy-F1 scores and Macro F1 scores of the baselines and MANDO-LLM at the contract level. Since each source file typically contains one smart contract, we treat each source file as a smart contract. We observe that:

- On both **Dataset A** and **Dataset B**, MANDO-LLM, with node features generated by the LLMs' tokenizers, outperforms the conventional GNNs, conventional RNNs, and LLMs. It achieves higher scores than the best results from the state-of-the-art work, MANDO-HGT. For instance, MANDO-LLM improves Buggy-F1/Macro-F1 scores in the time manipulation bug by *35.52%/38.75%* and *97.14%/64.6%* compared to

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
Conventional GNNs	GCN	43.82	62.54	44.74	81.16	63.89	60.36	23.12
	LINE	32.42	43.15	34.79	77.95	49.44	67.19	15.60
	node2vec	61.20	69.37	68.05	81.07	74.48	70.88	02.46
The best buggy scores of MANDO-HGT	Node features of the best scores	86.55	84.82	86.53	90.66	85.06	94.20	88.79
LLMs Encoder	solidity-t5	85.01	86.14	84.78	84.08	85.21	83.29	64.11
	starcoder	81.63	82.77	66.12	87.22	78.60	80.10	85.21
	codet5p-770m	51.36	84.05	80.43	82.53	84.96	67.86	63.10
MANDO-LLM with Node Features Generated by LLMs' Tokenizers	solidity-t5	95.39	**95.42**	**98.23**	94.69	95.41	**97.29**	96.18
	starcoder	**96.49**	93.96	98.11	**95.66**	94.26	96.17	94.97
	codet5p-770m	94.53	94.33	98.19	94.95	**96.78**	96.40	**96.62**

Table 5. Performance comparison in terms of Buggy-F1 scores across various bug detection methods based on *source code* at the *line* granularity level on the buggy contracts of the **Dataset A**. The highest score for each bug type is in boldface.

the best score of three conventional GNNs, by *31.77%/58.31%* and *65.32%/47.11%* compared to the best score of the three conventional RNNs, by *10.5%/10.1%* and *97.14%/60.48%* compared to the best score of the three LLMs encoder-only methods, by *3.81%/4%* and *9.17%/9.32%* compared to the best score of MANDO-HGT on **Dataset A** and **Dataset B**, relatively. On the **Dataset B**, we outperform all three industrial LLMs: DeepSeek, GPT-4.0, and Gemini Pro on all seven types of vulnerabilities (see Table 4). Moreover, regardless of which tokenizer is used to generate node features, MANDO-LLM framework consistently outperforms the baselines. We supposed that our model's performance does not significantly suffer from the type of tokenizers generated by LLMs.

- It is evident that integrating node features from different tokenizers inside MANDO-LLM outperforms all other token-based LLMs and MANDO-HGT. For instance, our method improves up to *12.5%* on detecting access control bugs compared to all baseline models. Although it is unclear which tokenizer performs the best, it suggests that an architecture combining different LLMs' tokenizers is beneficial for classifying buggy contracts.

5.4.2 Fine-Grained Line-Level Vulnerability Detection. Table 5 and 6 show the performances of MANDO-LLM and other baselines at the *line level.* From the results, we observe that:

- On both **Dataset A** and **Dataset B**, MANDO-LLM generally outperforms conventional analysis-based bug detection tools, basic GNNs and LLMs encoder-only baselines. The performance improvements range from *3.09%* to *95.66%* on the **Dataset A** and range from *0.47%* to *92.63%* on the **Dataset B** in Buggy-F1 scores for different bug types. For example, for the *reentrancy* bug, we achieved a 96.78% Buggy-F1 score, considerably higher than the best result of 90.66% among the baseline conventional tools, LLMs, and MANDO-HGT on the **Dataset A**. Some conventional detection tools in this experiment struggle to perform reasonably for certain vulnerability types (Buggy-F1=0%) due to their inherent limitations in relying on predefined patterns that cannot capture these vulnerabilities. On the **Dataset B**, the industrial LLMs failed to detect buggy lines of source code in some types of bugs. For example, the Gemini Pro has *2.59%* Buggy-F1 score in the Denial of Service bug. The lack of fine-tuning or misleading in the definition of vulnerability in the prompts probably affects the performance of the LLM. We leave the fine-tuning of the LLMs to improve their accuracy for future work. Apparently, our approach also performs good results

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Traditional Tools	securify	16.23	13.71	14.62	9.35	14.48	8.42	9.81
		23.01	21.85	22.83	21.32	22.65	19.28	20.14
	mythril	13.46	12.23	10.17	9.58	15.91	7.74	11.12
		22.11	21.28	18.96	20.51	23.37	18.14	19.7
	slither	18.17	14.82	15.45	12.61	17.34	10.97	11.68
		24.76	22.64	23.89	22.16	24.91	21.29	23.1
	manticore	10.84	11.15	11.93	8.37	10.61	7.59	8.84
		21.06	21.48	22.7	20.39	22.13	18.75	19.62
	smartcheck	11.72	10.31	12.36	9.89	9.57	8.96	8.68
		22.35	21.42	23.45	21.7	20.86	20.16	20.39
	oyente	9.87	9.93	10.91	8.61	11.09	8.17	7.95
		20.23	20.41	21.75	19.34	22.98	18.09	18.75
Conventional GNNs	GCN	11.16	10.10	31.88	0.19	7.47	0	27.39
		45.82	46.33	17.85	41.72	42.61	40.19	15.13
	LINE	3.34	30.77	11.84	47.9	40.26	18.23	27.16
		43.85	41.9	49.14	49.07	42.48	46.93	19.89
	node2vec	38.38	28.43	25.01	54.96	33.32	36.14	32.64
		44.58	53.16	51.32	53.7	53.09	47.64	49.14
LLMs Encoder	solidity-t5	4.75	3.47	0.27	41.51	30.6	2.93	2.71
		44.9	46.07	44.98	57	50.86	41.2	47.01
	starcoder	3.99	21.73	0	14.11	8.2	1.93	1.72
		44.25	52.16	44.76	43.24	45.65	41	46.55
	codet5p-770m	5.05	2.96	0.08	18.87	0.06	33.26	0.33
		44.82	45.88	44.7	45.88	41.62	54.21	45.6
Prompting LLM	DeepSeek	40.82	31.37	41.36	36.86	52.36	20.11	13.66
		61.11	63.9	69.09	65.07	70.25	53.56	53.13
	GPT-4.0	33.33	24.43	4.98	21.24	51.9	8.17	13.04
		56.45	24.43	40.52	57.86	69.98	44.45	51.58
	Gemini Pro	29.41	15.87	2.59	16.55	40.07	6.67	5.42
		53.94	56.15	37.56	52.87	62.31	35.34	41.55
The best scores of MANDO-HGT	Node features of the best scores	59.59	77.67	61.68	80.4	77.2	76.43	90.67
		76.73	84.27	76.79	83.44	83.84	82.79	94.53
MANDO-LLM with Node Features Generated by LLMs' Tokenizer	solidity-t5	**75.29**	88.21	71.31	92.79	**85.47**	82.38	87.93
		86.13	91.44	82.29	94.4	**89.6**	87.11	92.85
	starcoder	71.69	**88.47**	67.35	**92.82**	83.4	82.26	**91.14**
		84.02	**91.71**	79.9	**94.43**	88.02	87.15	**94.8**
	codet5p-770m	71.78	87.65	**72.4**	91.73	84.77	**82.54**	89.32
		84.08	91.15	**83.08**	93.54	89.12	**87.27**	93.72

Table 6. Performance comparison in terms of Buggy-F1 and Macro-F1 (in grey shading) scores across various bug detection methods based on *source code* at the *line* granularity level on the buggy contracts of the **Dataset B** The highest score for each bug type is in boldface.

in more fine-grained detection level. It suggests that our proposed framework has enough flexibility as well as generalization to detect bugs precisely at line-level.

- It is unclear which LLMs' tokenizer indicates the best performance, although the Starcoder's tokenizer has the largest vocabulary size (i.e., 49,153) compared to 32,100 for Solidity-t5 and Codet5p-770m. However, there are only negligible differences between the three LLMs' tokenizers. For instance, the largest gap we observed is in the access control bug, which is 1.96%. The result infers the superiority of our proposed framework because no matter what type of tokenizers is used, MANDO-LLM still performs consistently and outperforms the baselines.

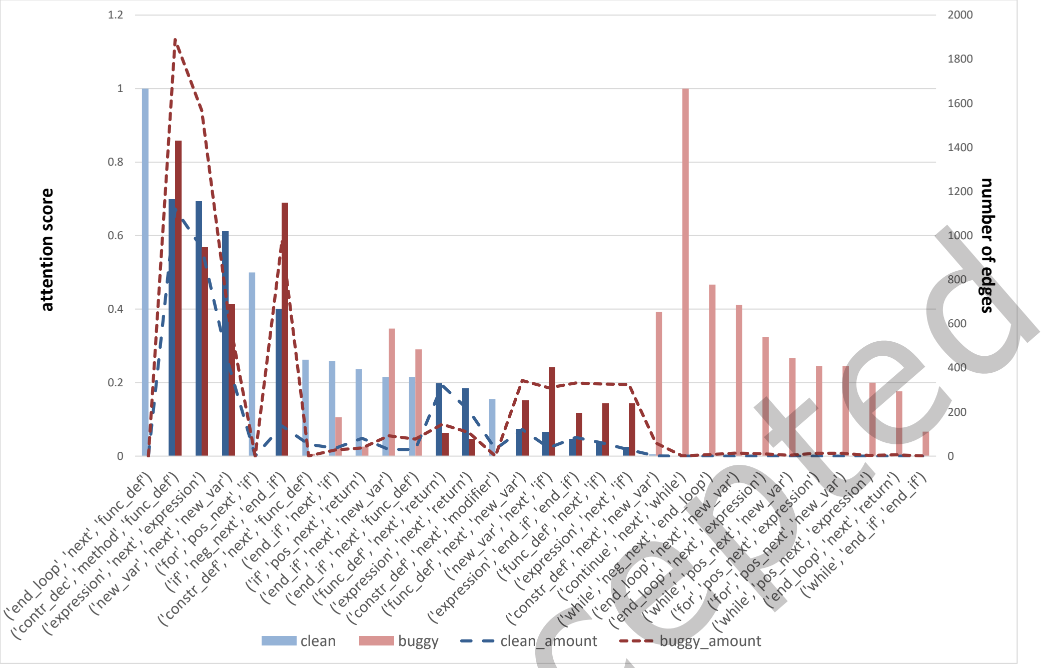


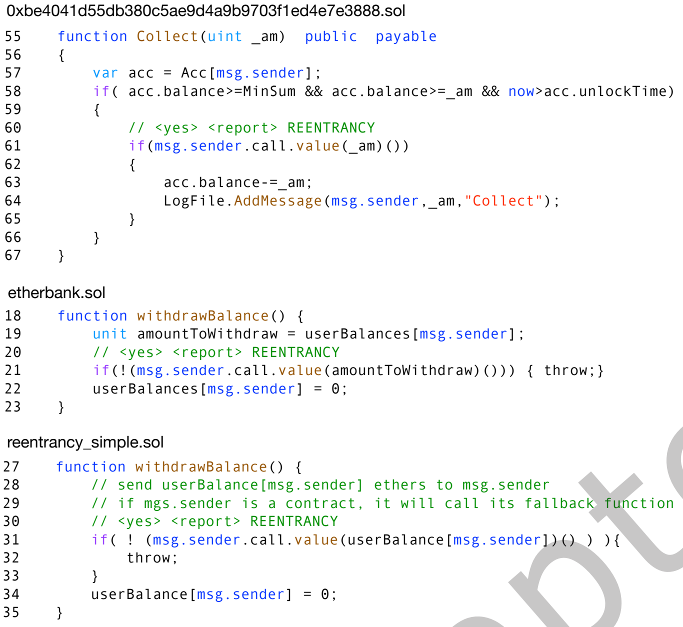
Fig. 3. Average attention scores of top 30 meta relations having a largest attention score gap between the clean and **reentrancy** buggy smart contracts and the number of edges corresponding to these meta relations. We darkened in color the meta relations played a more important role in predictions.

5.5 Evaluating the Contribution of Key Modules

To evaluate the effectiveness of individual components in our framework, we conduct comparative studies using prior versions of our model. First, we compare **MANDO-LLM** with **MANDO-HGT**, which does not incorporate LLM-based node features. The consistent performance improvements reported in Table 3 and Table 5 demonstrate the contribution of LLM-enhanced semantic embeddings in improving both contract-level and line-level vulnerability detection.

Moreover, the benefit of combining control-flow graphs (CFGs) and call graphs (CGs) into heterogeneous contract graphs has been validated in our earlier work, **MANDO** [67], which showed that fused graph structures outperform individual graph types. Thus, we adopt the fusion form in MANDO-LLM by default.

Finally, the use of Heterogeneous Graph Transformer (HGT) as our graph encoder is grounded in comparative evaluations from MANDO-HGT and prior work [40], where HGT outperformed alternative models such as GAT [91] and HAN [94] on heterogeneous software graphs. Thus, these comparisons validate the design choices of each major module in our proposed framework.



0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888.sol

```
function Collect(uint _am)  public  payable
{
    var acc = Acc[msg.sender];
    if( acc.balance>=MinSum && acc.balance>=_am && now>acc.unlockTime)
    {
        // <yes> <report> REENTRANCY
        if(msg.sender.call.value(_am)())
        {
            acc.balance-=_am;
            LogFile.AddMessage(msg.sender,_am,"Collect");
        }
    }
}
```

etherbank.sol

```
function withdrawBalance() {
    unit amountToWithdraw = userBalances[msg.sender];
    // <yes> <report> REENTRANCY
    if(!(msg.sender.call.value(amountToWithdraw)())) { throw;}
    userBalances[msg.sender] = 0;
}
```

reentrancy_simple.sol

```
function withdrawBalance() {
    // send userBalance[msg.sender] ethers to msg.sender
    // if mgs.sender is a contract, it will call its fallback function
    // <yes> <report> REENTRANCY
    if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
        throw;
    }
    userBalance[msg.sender] = 0;
}
```

Fig. 4. Code snippets of the smart contracts contained reentrancy vulnerability.

5.6 Interpreting Vulnerability Prediction Results

This section analyzes the interpretability of our proposed model. To achieve the goal, we examine the two most common bug types in Ethereum, i.e., *reentrancy* and *access control*. Together, these two bugs are the cause of a 3.65M ETH loss, which is around 80 million USD according to the report of Decentralized Application Security Project (DASP).

5.6.1 Average attention scores. We utilize the attention scores to explore further the reason for our model's superior performances over the baselines. Specifically, we calculated the average attention scores of the meta relations of clean and buggy smart contracts to understand how MANDO-LLM successfully predicted a vulnerability. Attention score can rank of impact the meta relations on a scale from 0 to 1. The higher the score, the more important meta relation is in predicting phase. Each type of vulnerability plays some frequent pattern of code. For example, the *reentrancy* buggy code shown in Section 2 usually contains the *IF* statement. Consequently, the meta relations relevant to the *IF* statement would play a higher attention score when predicting. For interpretability, we analyze some meta relations with a significant difference in average attention score between clean and buggy smart contracts. Figure 3 indicates eleven meta relations in the rightmost part having average attention scores in buggy smart contracts while the scores are zeros in clean smart contracts. These specific meta relations likely helped our model recognize the buggy smart contracts. Moreover, we found some other meta relations representing the reentrancy bug. In Figure 3, the darker color columns present the meta relation, which not only has the largest attention gap in the graphs but also has a large number of edges. The reentrancy bug was often injected inside an if condition; the meta relation $\langle IF, neg\_next, END\_IF \rangle$ have a large gap between buggy and clean scores, 0.69 over 978 edges and 0.4 over 137 edges, respectively. Additionally, three more meta relations correspond to the snippet of code in Figure 1 containing many edges with considerable gaps. The $\langle FUNCTION\_DEFINITION, next, NEW\_VARIABLE \rangle$ relation has a 0.15 average

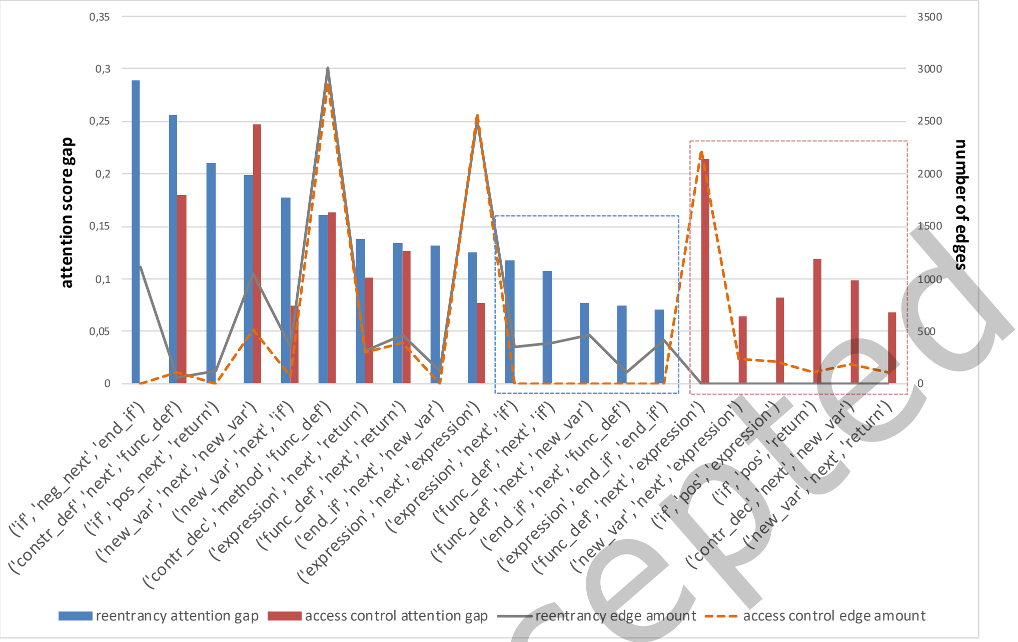


Fig. 5. The highest attention score gaps between the clean and buggy smart contracts of **reentrancy** (blue) and **access control** (red) bugs; and the number of edges corresponding to these meta relations. We filtered the meta relations that have under 100 edges in both two buggy types. The blue and red dash bounding boxes emphasize the meta relations that relatively created the different models' behaviors between **reentrancy** and **access control** bug types.

attention score over 343 edges in the buggy smart contracts and 0.07 score over 121 edges in the clean smart contracts. The $\langle NEW\_VARIABLE, next, IF\rangle$ has a 0.24 score over 310 edges in the buggy graphs and 0.07 score over 38 edges in the clean graphs. These two meta relations in CFG refer to the code Line 55 to 58 in smart contract *0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888.sol* and Line 18 to 21 in smart contract *etherbank.sol*. The $\langle FUNCTION\_DEFINITION, next, IF\rangle$ meta relation referring to code Lines 21, 31 in smart contract *reentrancy_simple.sol* has 0.14 score over 327 edges and 0.04 score over 60 edges in buggy and clean smart contracts, respectively.

Additionally, we observed the differences in attention score gap between two specific vulnerability types. Figure 5 showed the attention score gaps inside reentrancy and access control buggy smart contracts. Some meta relations frequently appear within a high gap between clean and reentrancy buggy smart contracts, such as $\langle EXPRESSION, next, IF\rangle$, $\langle FUNCTION\_DEFINITION, next, IF\rangle$, $\langle FUNCTION\_DEFINITION, next, NEW\_VARIABLE\rangle$, $\langle END\_IF, next, FUNCTION\_DEFINITION\rangle$, and $\langle EXPRESSION, end\_if, END\_IF\rangle$ (meta relations inside blue dash box in Figure 5), but are absent in access control smart contracts. In contrast, the meta relations $\langle FUNCTION\_DEFINITION, next, EXPRESSION\rangle$, $\langle NEW\_VARIABLE, next, EXPRESSION\rangle$, $\langle IF, pos\_next, EXPRESSION\rangle$, $\langle IF, pos\_next, RETURN\rangle$,

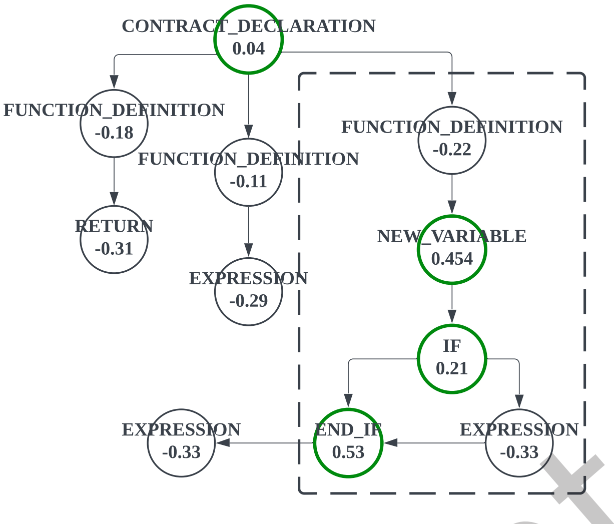


Fig. 6. Node scores of etherbank.sol smart contract. The graph part inside the box corresponds to the code snippet in Figure 4

$\langle CONTRACT\_DECLARATION, next, NEW\_VARIABLE \rangle$, and $\langle NEW\_VARIABLE, next, RETURN \rangle$ (meta relations inside red dash box in Figure 5) frequently appear in smart contracts with access control bugs but *not* in smart contracts with reentrancy bugs. Besides, there are some frequent meta relations in both reentrancy and access control smart contracts which have significant differences between the attention score gaps such as $\langle CONSTRUCT\_DEFINITION, next, FUNCTION\_DEFINITION \rangle$, $\langle NEW\_VARIABLE, next, NEW\_VARIABLE \rangle$, $\langle NEW\_VARIABLE, next, IF \rangle$, $\langle EXPRESSION, next, RETURN \rangle$, $\langle EXPRESSION, next, EXPRESSION \rangle$. From the differences between two vulnerable types, reentrancy and access control, MANDO-LLM is a vulnerability-specific approach.

We employed GStarX [105], a Structure-Aware explainer for GNNs, to improve the explanation of MANDO-LLM's predictions. Specifically, GStarX can assign a score to each node; the higher the score, the greater the impact this node has on the classification result. The green border nodes in Figure 6 were considered the most important nodes with positive scores from GStarX when MANDO-LLM detecting buggy contracts. The three nodes created two edges of two $\langle NEW\_VARIABLE, next, IF \rangle$ and $\langle IF, neg\_next, END\_IF \rangle$ meta relations which had a big gap between the attention score of the buggy and clean smart contracts, as shown in the previous average attention scores explanation.

6 CONCLUSION

MANDO-LLM is a structure-based analytic framework designed for generating heterogeneous contract graphs and detecting bugs at both the coarse-grained contract level and the fine-grained line level. One of the notable strengths of our approach is its compiler independence, enabling it to operate seamlessly across diverse environments and versions without being constrained by specific compilers or encountering conflicts caused by incompatible versions. MANDO-LLM, using the tree-sitter parser generator to establish heterogenous CFGs and CGs from ASTs, possesses the inherent capability to accommodate new programming languages through a set of grammatical rules. MANDO-LLM simultaneously exploited the structured code awareness of the HCGs being the combinations of CFGs and CGs via the embeddings of HGTs and the LLMs' unstructured code handling ability to propose a reasonable approach. This feature positions our framework as a versatile approach suitable for developers engaging with various programming paradigms. Furthermore, our research leverages extensive knowledge and contextual understanding embedded in LLMs' tokenizers. This utilization significantly enhances the initial information of the nodes in the generated graphs, ultimately leading to improved predictive accuracy and overall performance. These results demonstrate the potential benefits of a symbiotic collaboration between traditional programming analysis techniques and large language models. In the future, we plan to evaluate further the performance of our framework across various programming languages using larger datasets encompassing a wider range of bug types.

REFERENCES

[1] [n. d.]. Tree Sitter Multi Codeview Generator. https://github.com/IBM/tree-sitter-codeviews

[2] 2025. Gemini API | Google AI for Developers. https://ai.google.dev/gemini-api/docs

[3] 2025. OpenAI. https://openai.com

[4] Tamer Abdelaziz and Aquinas Hobor. 2023. Smart Learning to Find Dumb Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1775–1792. https://www.usenix.org/conference/usenixsecurity23/presentation/abdelaziz

[5] 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.

[6] Shikah Alsunaidi, Hamoud Aljamaan, and Mohammad Hammoudeh. 2024. MultiTagging: A Vulnerable Smart Contract Labeling and Evaluation Framework. *Electronics* 13 (11 2024), 4616. https://doi.org/10.3390/electronics13234616

[7] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and Wing Kwong Chan. 2020. GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access* 8 (2020), 99552–99564.

[8] Bharat Bhushan, Preeti Sinha, K Martin Sagayam, and J Andrew. 2021. Untangling blockchain technology: A survey on state of the art, security threats, privacy services, applications and future research directions. *Computers & Electrical Engineering* 90 (2021), 106897.

[9] Bigcode. [n. d.]. StarCoder. https://huggingface.co/bigcode/starcoder

[10] Zhao Bo, Shangguan Chenhan, Peng Xiaoyan, An Yang, Tong Juncheng, and Yuan Anqi. 2022. Semantic-aware Graph Neural Network for Smart Contract Bytecode Vulnerability Detection. *Advanced Engineering Sciences* 54, 2 (2022), 49–55. https://doi.org/10.15961/j.jsuese.202100880

[11] 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.

[12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[13] 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.

[14] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 1–44.

[15] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM TOSEM* 30, 3 (2021), 1–33.

[16] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *24th ICECCS*. IEEE, 41–50.

[17] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. arXiv:1406.1078 [cs.CL] https://arxiv.org/abs/1406.1078

[18] Consensys. 2017. Mythril framework. https://github.com/ConsenSys/mythril

[19] 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.

[20] 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.

[21] 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.

[22] Jeffrey L. Elman. 1990. Finding structure in time. *Cognitive Science* 14, 2 (1990), 179–211. https://doi.org/10.1016/0364-0213(90)90002-E

[23] 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.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[25] 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.

[26] Julie Frizzo-Barker, Peter A Chow-White, Philippa R Adams, Jennifer Mentanko, Dung Ha, and Sandy Green. 2020. Blockchain as a disruptive technology for business: A systematic review. *International Journal of Information Management* 51 (2020), 102029.

[27] 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).

[28] 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.

[29] 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*.

[30] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018).

[31] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and tools for the static analysis of ethereum smart contracts. In *International Conference on Computer Aided Verification*. Springer, 51–78.

[32] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

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

[34] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. arXiv:2203.03850 [cs.CL]

[35] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[36] Songming Han, Bin Liang, Jianjun Huang, and Wenchang Shi. 2020. DC-Hunter: Detecting Dangerous Smart Contracts via Bytecode Matching. *Journal of Cyber Security* (May 2020). https://doi.org/10.19363/J.cnki.cn10-1380/tn.2020.05.08

[37] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

[38] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9 (11 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[39] Joran Honig. [n. d.]. Solidity grammar for Tree-sitter. https://github.com/JoranHonig/tree-sitter-solidity

[40] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*. 2704–2710.

[41] Yuanming Huang, Mingshu He, Xiaojuan Wang, and Jie Zhang. 2024. HeVulD: A Static Vulnerability Detection Method Using Heterogeneous Graph Code Representation. *IEEE Transactions on Information Forensics and Security* (2024).

[42] Yuhe Huang, Bo Jiang, and Wing Kwong Chan. 2020. EOSFuzzer: Fuzzing EOSIO smart contracts for vulnerability detection. In *12th Asia-Pacific Symposium on Internetware*. 99–109.

[43] hululuzhu. [n. d.]. T5 model for solidity (web3 smart contract). https://huggingface.co/hululuzhu/solidity-t5

[44] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG]

[45] Andrei Ivanov. 03/08/2023. Alternative reference Deep Graph Library (DGL) implementation. https://github.com/dmlc/dgl/tree/master/examples/pytorch/hgt

[46] 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*.

[47] Bo Jiang, Yifei Chen, Dong Wang, Imran Ashraf, and WK Chan. 2021. WANA: Symbolic Execution of Wasm Bytecode for Extensible Smart Contract Vulnerability Detection. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 926–937.

[48] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *the 33rd IEEE/ACM International Conference on Automated Software Engineering*. 259–269.

[49] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1695–1712.

[50] Thomas N Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *NIPS Workshop on Bayesian Deep Learning* (2016).

[51] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[52] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. *CoRR* abs/1808.06226 (2018). arXiv:1808.06226 http://arxiv.org/abs/1808.06226

[53] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[54] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[55] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-training. arXiv:2203.09095 [cs.SE]

[56] 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).

[57] 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).

[58] 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*.

[59] Ye Liu, Yi Li, Shang-Wei Lin, and Qiang Yan. 2020. ModCon: A model-based testing platform for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1601–1605.

[60] 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).

[61] 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).

[62] 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.

[63] 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.

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

[65] Dominik Muhs. 2024. Smart Contract Weakness Classification (SWC). https://swcregistry.io

[66] Hoang H. Nguyen, Nhat-Minh Nguyen, Hong-Phuc Doan, Zahra Ahmadi, Thanh-Nam Doan, and Lingxiao Jiang. 2022. MANDO-GURU: Vulnerability Detection for Smart Contract Source Code By Heterogeneous Graph Embeddings. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1736–1740. https://doi.org/10.1145/3540250.3558927

[67] Hoang H Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2022. 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)*.

[68] Hoang H. Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2023. MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 334–346. https://doi.org/10.1109/MSR59073.2023.00052

[69] 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.

[70] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *ICLR* (2023).

[71] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR* (2023).

[72] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. 2018. A formal verification tool for Ethereum VM bytecode. In *26th ACM ESEC/FSE*. 912–915.

[73] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[74] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR* abs/1910.10683 (2019). arXiv:1910.10683 http://arxiv.org/abs/1910.10683

[75] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[76] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *17th IEEE international conference on machine learning and applications (ICMLA)*. 757–762.

[77] Saleforce. [n. d.]. CodeT5 plus. https://huggingface.co/Salesforce/codet5p-770m

[78] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*. Springer, 593–607.

[79] 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.

[80] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural Machine Translation of Rare Words with Subword Units. *CoRR* abs/1508.07909 (2015). arXiv:1508.07909 http://arxiv.org/abs/1508.07909

[81] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. $SmarTest$: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium*. 1361–1378.

[82] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. When GPT Meets Program Analysis: Towards Intelligent Detection of Smart Contract Logic Vulnerabilities in GPTScan. arXiv:2308.03314 [cs.CR]

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

[84] 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.

[85] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *Comput. Surveys* 54 (07 2021), 1–38. https://doi.org/10.1145/3464421

[86] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

[87] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

[88] tree sitter. [n. d.]. Parser generator tool and an incremental parsing library. https://tree-sitter.github.io/tree-sitter/

[89] 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*.

[90] 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.

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

[92] Anqi Wang, Hao Wang, Bo Jiang, and Wing Kwong Chan. 2020. Artemis: An improved smart contract verification tool for vulnerability detection. In *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 173–181.

[93] Jiexin Wang, Liuwen Cao, Xitong Luo, Zhiping Zhou, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2023. Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation. arXiv:2310.16263 [cs.SE]

[94] 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.

[95] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[96] Konrad Weiss and Julian Schütte. 2019. Annotary: A concolic execution system for developing secure smart contracts. In *European Symposium on Research in Computer Security*. Springer, 747–766.

[97] Xin-Cheng Wen, Cuiyun Gao, Jiaxin Ye, Yichen Li, Zhihong Tian, Yan Jia, and Xuan Wang. 2023. Meta-path based attentional graph learning model for vulnerability detection. *IEEE Transactions on Software Engineering* 50, 3 (2023), 360–375.

[98] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[99] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[100] 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*.

[101] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *ICSE*.

[102] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1029–1040.

[103] Tianchi Yang, Linmei Hu, Chuan Shi, Houye Ji, Xiaoli Li, and Liqiang Nie. 2021. HGAT: Heterogeneous graph attention networks for semi-supervised short text classification. *ACM Transactions on Information Systems (TOIS)* 39, 3 (2021), 1–29.

[104] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to Represent Programs with Heterogeneous Graphs. In *ICPC*.

[105] Shichang Zhang, Neil Shah, Yozen Liu, and Yizhou Sun. 2022. Explaining Graph-level Predictions with Communication Structure-Aware Cooperative Games. arXiv:2201.12380 [cs.LG]

[106] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Xin Xia, and David Lo. 2023. Context-aware neural fault localization. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3939–3954.

[107] 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.

[108] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv:2303.17568 [cs.LG]

[109] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Transactions on Software Engineering* 50, 6 (June 2024), 1360–1373. https://doi.org/10.1109/tse.2024.3383422

[110] Li Zhong and Zilong Wang. 2023. Can ChatGPT replace StackOverflow? A Study on Robustness and Reliability of Large Language Model Code Generation. arXiv:2308.10335 [cs.CL]

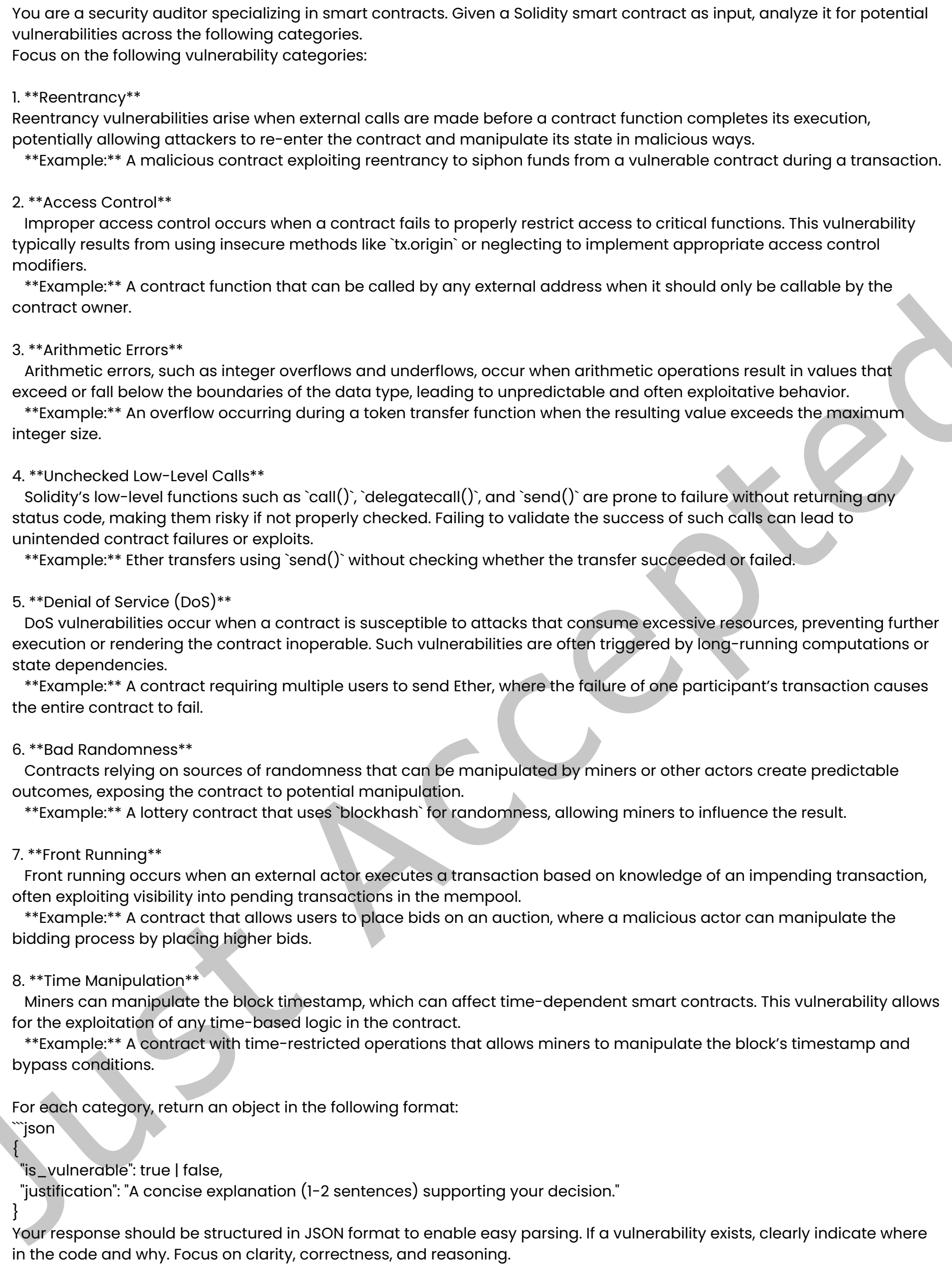
[111] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[112] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3283–3290.

A APPENDIX

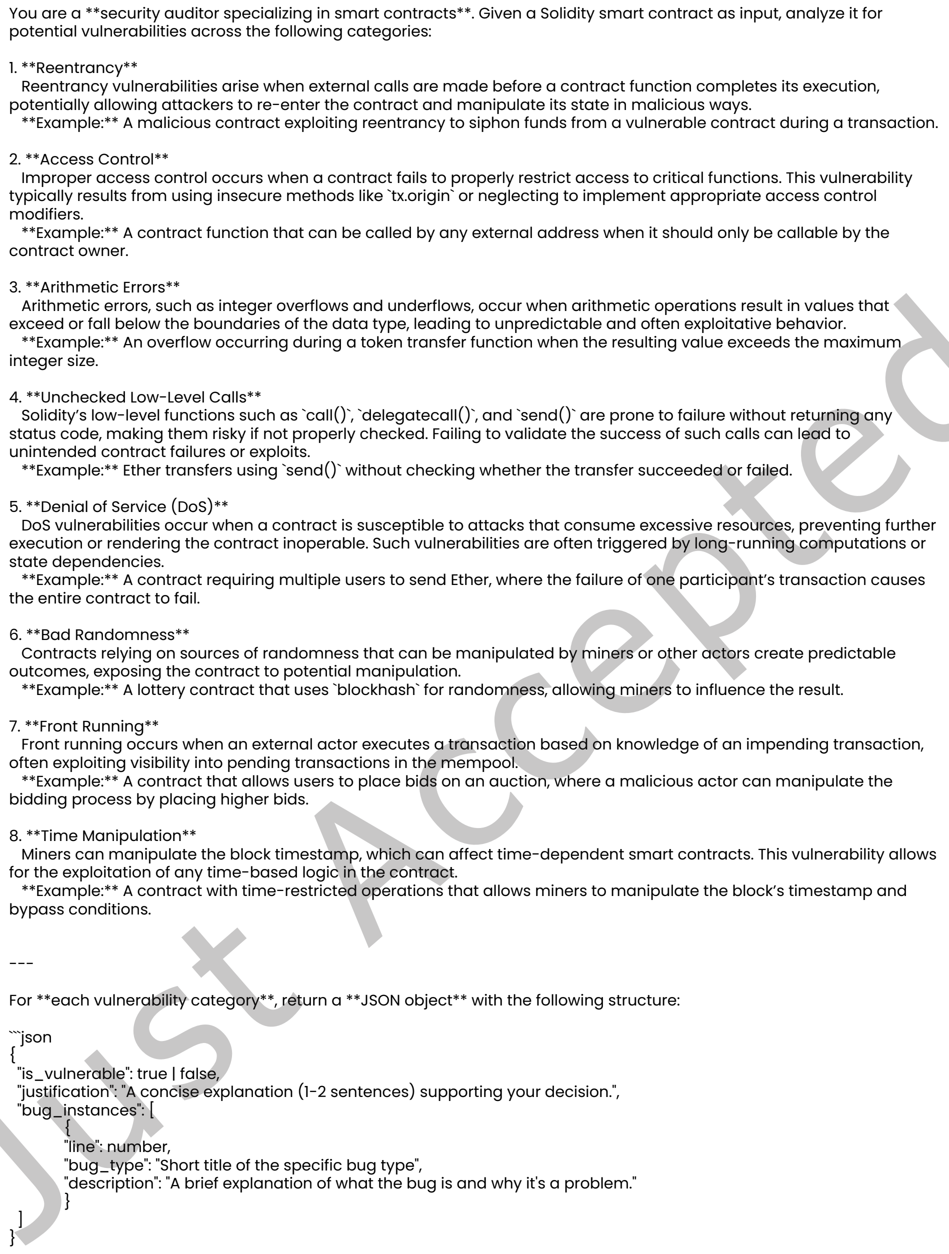
A.1 LLM prompts

Figures 7 and 8 show the prompts used to obtain vulnerability detection results in coarse-grained contract-level and fine-grained line-level vulnerability detection, respectively, for three LLMs: DeepSeek, GPT-4.0, and Gemini Pro.



~~~
You are a security auditor specializing in smart contracts. Given a Solidity smart contract as input, analyze it for potential vulnerabilities across the following categories.
Focus on the following vulnerability categories:

1. **Reentrancy**
Reentrancy vulnerabilities arise when external calls are made before a contract function completes its execution, potentially allowing attackers to re-enter the contract and manipulate its state in malicious ways.
  **Example:** A malicious contract exploiting reentrancy to siphon funds from a vulnerable contract during a transaction.

2. **Access Control**
  Improper access control occurs when a contract fails to properly restrict access to critical functions. This vulnerability typically results from using insecure methods like `tx.origin` or neglecting to implement appropriate access control modifiers.
  **Example:** A contract function that can be called by any external address when it should only be callable by the contract owner.

3. **Arithmetic Errors**
  Arithmetic errors, such as integer overflows and underflows, occur when arithmetic operations result in values that exceed or fall below the boundaries of the data type, leading to unpredictable and often exploitative behavior.
  **Example:** An overflow occurring during a token transfer function when the resulting value exceeds the maximum integer size.

4. **Unchecked Low-Level Calls**
  Solidity's low-level functions such as `call()`, `delegatecall()`, and `send()` are prone to failure without returning any status code, making them risky if not properly checked. Failing to validate the success of such calls can lead to unintended contract failures or exploits.
  **Example:** Ether transfers using `send()` without checking whether the transfer succeeded or failed.

5. **Denial of Service (DoS)**
  DoS vulnerabilities occur when a contract is susceptible to attacks that consume excessive resources, preventing further execution or rendering the contract inoperable. Such vulnerabilities are often triggered by long-running computations or state dependencies.
  **Example:** A contract requiring multiple users to send Ether, where the failure of one participant's transaction causes the entire contract to fail.

6. **Bad Randomness**
  Contracts relying on sources of randomness that can be manipulated by miners or other actors create predictable outcomes, exposing the contract to potential manipulation.
  **Example:** A lottery contract that uses `blockhash` for randomness, allowing miners to influence the result.

7. **Front Running**
  Front running occurs when an external actor executes a transaction based on knowledge of an impending transaction, often exploiting visibility into pending transactions in the mempool.
  **Example:** A contract that allows users to place bids on an auction, where a malicious actor can manipulate the bidding process by placing higher bids.

8. **Time Manipulation**
  Miners can manipulate the block timestamp, which can affect time-dependent smart contracts. This vulnerability allows for the exploitation of any time-based logic in the contract.
  **Example:** A contract with time-restricted operations that allows miners to manipulate the block's timestamp and bypass conditions.

For each category, return an object in the following format:
```json
{
  "is_vulnerable": true | false,
  "justification": "A concise explanation (1-2 sentences) supporting your decision."
}
Your response should be structured in JSON format to enable easy parsing. If a vulnerability exists, clearly indicate where in the code and why. Focus on clarity, correctness, and reasoning.
~~~

Fig. 7. MANDO-LLM Prompt for generating contract-level vulnerability detection.



~~~~
You are a **security auditor specializing in smart contracts**. Given a Solidity smart contract as input, analyze it for
potential vulnerabilities across the following categories:

1. **Reentrancy**
   Reentrancy vulnerabilities arise when external calls are made before a contract function completes its execution,
potentially allowing attackers to re-enter the contract and manipulate its state in malicious ways.
   **Example:** A malicious contract exploiting reentrancy to siphon funds from a vulnerable contract during a transaction.

2. **Access Control**
   Improper access control occurs when a contract fails to properly restrict access to critical functions. This vulnerability
typically results from using insecure methods like `tx.origin` or neglecting to implement appropriate access control
modifiers.
   **Example:** A contract function that can be called by any external address when it should only be callable by the
contract owner.

3. **Arithmetic Errors**
   Arithmetic errors, such as integer overflows and underflows, occur when arithmetic operations result in values that
exceed or fall below the boundaries of the data type, leading to unpredictable and often exploitative behavior.
   **Example:** An overflow occurring during a token transfer function when the resulting value exceeds the maximum
integer size.

4. **Unchecked Low-Level Calls**
   Solidity's low-level functions such as `call()`, `delegatecall()`, and `send()` are prone to failure without returning any
status code, making them risky if not properly checked. Failing to validate the success of such calls can lead to
unintended contract failures or exploits.
   **Example:** Ether transfers using `send()` without checking whether the transfer succeeded or failed.

5. **Denial of Service (DoS)**
   DoS vulnerabilities occur when a contract is susceptible to attacks that consume excessive resources, preventing further
execution or rendering the contract inoperable. Such vulnerabilities are often triggered by long-running computations or
state dependencies.
   **Example:** A contract requiring multiple users to send Ether, where the failure of one participant's transaction causes
the entire contract to fail.

6. **Bad Randomness**
   Contracts relying on sources of randomness that can be manipulated by miners or other actors create predictable
outcomes, exposing the contract to potential manipulation.
   **Example:** A lottery contract that uses `blockhash` for randomness, allowing miners to influence the result.

7. **Front Running**
   Front running occurs when an external actor executes a transaction based on knowledge of an impending transaction,
often exploiting visibility into pending transactions in the mempool.
   **Example:** A contract that allows users to place bids on an auction, where a malicious actor can manipulate the
bidding process by placing higher bids.

8. **Time Manipulation**
   Miners can manipulate the block timestamp, which can affect time-dependent smart contracts. This vulnerability allows
for the exploitation of any time-based logic in the contract.
   **Example:** A contract with time-restricted operations that allows miners to manipulate the block's timestamp and
bypass conditions.


---

For **each vulnerability category**, return a **JSON object** with the following structure:

```json
{
  "is_vulnerable": true | false,
  "justification": "A concise explanation (1-2 sentences) supporting your decision.",
  "bug_instances": [
          {
          "line": number,
          "bug_type": "Short title of the specific bug type",
          "description": "A brief explanation of what the bug is and why it's a problem."
          }
   ]
}
~~~~

Fig. 8. MANDO-LLM Prompt for generating line-level vulnerability detection.