

**TITLE OF INVENTION: ELECTRONIC PLATFORM FOR DETECTING BUGS IN SMART
CONTRACT CODE**

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] The present application claims priority from Singapore Patent Application No. 10202500228T titled “ELECTRONIC PLATFORM FOR DETECTING BUGS IN SMART CONTRACT CODE” and filed on 24 January 2025, the content of which is hereby incorporated by reference in its entirety.

FIELD

[0002] The present application relates to computational methods an electronic platform for detecting bugs in smart contract code. In particular, the application relates to methods and electronic platforms that represent smart contract source code as heterogeneous program graphs and apply graph-based machine learning models augmented with language-model-derived code representations to identify vulnerabilities at both coarse-grained and fine-grained levels.

BACKGROUND

[0003] Blockchain-based smart contracts have found applications across various domains, including finance, e-commerce, healthcare, logistics, and law. Smart contracts are software code which deployed and then is auto executed on a blockchain, and as such the smart contract cannot be patched after deployment. Any bug or security vulnerability in a deployed smart contract can lead to severe consequences for both developers and users. As a result, there is a high demand for security assurance techniques, particularly those for smart contract vulnerabilities.

[0004] Existing methodologies 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.

[0005] Techniques based on machine learning and deep learning approaches seek to address these challenges by encoding various syntactic and semantic code information through syntax trees, control-flow graphs, or program dependency graphs. Automated classifiers are employed to differentiate vulnerable code from non-vulnerable ones. However, existing learning-based techniques typically rely on either graph-

based representations that emphasize structural relationships but lack awareness of global control-flow and inter-procedural relationships. As a result, such approaches struggle to accurately detect complex vulnerabilities that depend on both semantic code behavior and cross-function execution context, particularly at fine-grained locations within the source code. Learning-based techniques reduce the need for manually specified patterns or specifications, making it easier to adapt to new programming languages 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 their code 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. [0006] There is thus a need for improved computational methods and systems for detecting vulnerabilities in smart contracts, or to at least provide a useful alternative to existing methods and systems.

SUMMARY

[0007] Embodiments of the present application discloses computational methods and systems that harness improved learning-based techniques for smart contract vulnerability detection, including the use of LLM tokenizers.

[0008] According to a first aspect there is provided an electronic platform for detecting bugs in smart contract code, the electronic platform comprising:

- at least one memory; and

- at least one processor configured to execute instructions stored in the memory that configure the at least one processor to:

- process received smart contract code to generate a heterogeneous contract graph comprising a plurality of nodes and at least one edge, where each node represents a semantic or syntactical segment of the smart contract code and each edge represents a control flow from a first node to a second node, wherein the plurality of nodes comprises at least two node types, and the at least one edge comprises at least one edge type;

- generate a plurality of node attribute features for each node by processing each node with a tokenizer of a pretrained Large Language Model (LLM), wherein the pretrained LLM is a LLM trained on a dataset comprising source code of at least one programming language and comprises a predetermined vocabulary and an embedding matrix or layer, and processing each node with the tokenizer comprises generating one or more tokens for each node using the predetermined vocabulary of the pretrained LLM and returning a vector of the embeddings for each of the one or more tokens using the embedding matrix or layer;

extract a plurality of meta relations from the nodes, the edges or both defining how connected nodes interact wherein the meta relations are dynamically determined based on node types and edge types present in the heterogeneous contract graph;

process the plurality of meta relations and the vectors with a trained graph-based machine learning model, which was trained on a training dataset comprising meta relations and vectors of correct smart contract code and buggy smart contract code, and the trained graph-based machine learning model is configured to perform heterogeneous message passing and relation-aware aggregation across node types and edge types and to detect a presence of one or more bugs in the received smart contract code and for each bug estimate a location of the respective bug in the source code; and

return a detection result indicating if one or more bugs were detected in the source codes, and a location of the respective bug in the source code.

[0009] According to a second aspect there is provided a computational method for detecting bugs in smart contract code, the method comprising:

processing received smart contract code to generate a heterogeneous contract graph comprising a plurality of nodes and at least one edge, where each node represents a semantic or syntactical segment of the smart contract code and each edge represents a control flow from a first node to a second node, wherein the plurality of nodes comprises at least two node types, and the at least one edge comprises at least one edge type;

generate a plurality of node attribute features for each node by processing each node with a tokenizer of a pretrained Large Language Model (LLM), wherein the pretrained LLM is a LLM trained on a dataset comprising source code of at least one programming language and comprises a predetermined vocabulary and an embedding matrix or layer, and processing each node with the tokenizer comprises generating one or more tokens for each node using the predetermined vocabulary of the pretrained LLM and returning a vector of the embeddings for each of the one or more tokens using the embedding matrix or layer;

extracting a plurality of meta relations from the nodes, the edges or both defining how connected nodes interact wherein the meta relations are dynamically determined based on node types and edge types present in the heterogeneous contract graph;

processing the plurality of meta relations and the vectors with a trained graph-based machine learning model, which was trained on a training dataset comprising meta relations and vectors of correct smart contract code and buggy smart contract code, and the trained machine learning model is configured to perform heterogeneous message passing and relation-aware aggregation across node types and edge types and to detect a presence of one or more bugs in the received smart contract code and for each bug estimate a location of the respective bug in the source code; and

returning a detection result indicating if one or more bugs were detected in the source codes, and a location of the respective bug in the source code.

[0010] According to a third aspect, there is provided a computer readable medium comprising instructions for configuring at least one processor to perform the method of the second aspect.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] Representative embodiments of the present disclosure are herein described, by way of example only, with reference to the accompanying drawings, wherein:

[0012] Figure 1A illustrates a snippet of a smart contract written in Solidity, featuring a reentrancy vulnerability and Figure 1B illustrates the call graph (CG) of the smart contract shown in Figure 1A and a portion of the control-flow graph (CFG) for the collect function within the given smart contract according to an embodiment.

[0013] Figure 2A is an architecture diagram showing the components of a smart contract vulnerability detector used by an electronic platform according to an embodiment and Figure 2B further illustrates an embodiment of a heterogeneous Graph Transformer Layer (HGT) layer shown in Figure 2A;

[0014] Figure 3 is a plot of the 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 from a interpretability analysis of an embodiment of a smart contract vulnerability detector where the slanted columns represent the attention scores for the clean smart contracts and the filled columns represent the attention scores for the reentrancy smart contracts, and the dashed line and solid line represent the number of edges for the clean and reentrancy smart contracts respectively, and the darker color columns present the meta relation which pay a more important role in predictions.

[0015] Figure 4 is a plot of the highest attention score gaps between the clean and buggy smart contracts from an interpretability analysis of an embodiment of a smart contract vulnerability detector where reentrancy bugs are shown in filled columns and access control bugs are shown in columns filled with slanted lines, along with the number of edges corresponding to these meta relations where the reentrancy edge amount is shown by the dashed line and the access control edge amount shown by the solid black line.

[0016] Figure 5 shows a node graph and node scores of the etherbank.sol smart contract from an interpretability analysis of an embodiment of a smart contract vulnerability detector where the part inside the box corresponds to the code snippet in Table 7 and the node scores for the most important nodes are shown within the bold dashed circles.

DETAILED DESCRIPTION

[0017] Embodiments of computational methods and electronic platforms for smart contract vulnerability detection will be described. These may implement a modular analysis pipeline that transforms smart contract code into a structured heterogeneous program representation, enriches such representation with semantic code features, and applies relation-aware graph learning to detect vulnerabilities. In the following description, various embodiments are described with reference to the drawings, where like reference characters generally refer to the same features across the drawings.

[0018] 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 in the present approach as heterogeneous graphs with various nodes and edges, like control flow and function-call graphs. This is illustrated in Figures 1A and 1B where Figure 1A illustrates a snippet of a smart contract written in Solidity, featuring a reentrancy vulnerability at line 11, and Figure 1B illustrates the call graph (CG) of the smart contract shown in Figure 1A and a portion of the control-flow graph (CFG) for the collect function within the given smart contract according to an embodiment.

[0019] To address existing graph learning techniques which struggle with large, complex graphs, embodiments the present approach 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. LLMs are used to capture code features from control-flow and call data. A customized HGT learns embeddings with specific node-edge meta relations and employs classifiers for vulnerability detection in smart contract code (in, for example, Solidity format) at both contract and line levels. Figure 2A, discussed in more detail below, is an architecture diagram showing the components of a smart contract vulnerability detector 100 used by an electronic platform 160 according to an embodiment. Figure 2B further illustrate an embodiment of a Heterogeneous Graph Transformer Layer (HGT) layer shown in Figure 2A. However, this is illustrative only, and other graph-based machine learning models and algorithms may also be used to process the meta relation and node features to identify vulnerabilities in smart contracts.

[0020] The present application takes advantage of LLMs containing 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 multilingual code generation models such as CodeGeeX, CodeGen, along with LLMs ability to handle unstructured code formats found in online forums or code repositories while ensuring code security. Specifically, Tokenizers 132 of pre-trained LLMs are used, which can effectively tokenize various programming languages, including Solidity source code.

[0021] The present approach features a syntax-based heterogeneous contract graph generator 110 supported by tree-sitter (although other similar programs could be used). This integration offers several advantages, including avoiding version conflicts and reducing the effort to set up entire projects due to

compiler and environment independence. The heterogeneous graph generator can easily adapt to other programming languages by providing a grammar set. This helps to enhance the information contained within the nodes of the generated heterogeneous contract graphs. LLM ability to handle effectively unstructured code format to embed the code snippet inside each node of the generated graphs is leveraged. 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 remain during LLM training. The present approach was found to outperform existing methods, with F1-score improvements from 0.59% to 80.72% at the contract level. An improvement in identifying line-level vulnerabilities is shown, with performance boosts ranging from 3.09% to over 95% across different vulnerability types, along with ease of retraining for various vulnerabilities without needing manually defined patterns.

[0022] Figure 1A depicts a snippet of a smart contract 10 written in Solidity, featuring a reentrancy vulnerability 11 at line 11. Figure 1B illustrates a Heterogeneous Contract Graph (HCG), fusing the heterogeneous call graph (CG) 20 of the smart contract 10, and a portion of the heterogeneous control-flow graph (CFG) 30 for the collect function within the given contract 10. 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 reentrancy 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, which can inadvertently trigger the smart contract's fallback function and make it vulnerable.

[0023] Prior art approaches which use predefined patterns may allow them to detect specific bugs, such as the reentrancy vulnerability and Time Manipulation. However, the present approach employs a heterogeneous graph structure, enabling a more general and flexible exploration of various vulnerability types without needing predefined patterns.

[0024] While heterogeneous control-flow graphs 30 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. Heterogeneous control-flow graphs typically comprise isolated sub-control-flow graphs of individual blocks. To overcome this constraint and enhance the understanding of the relationship among the code blocks inside a smart contract, the present approach integrates call graphs 20 into heterogeneous control-flow graphs when heterogeneous control-flow graphs assume a pivotal role by establishing linking edges 22h whenever one code block invokes other blocks or external libraries. This fusion of heterogeneous control-flow graphs 30 and heterogeneous call graphs 20 creates a composite graph, which we refer to as a

Heterogeneous Contract Graph (HCG) 118 that can effectively capture more complex code patterns that link up to other control flows outside the current block.

[0025] Heterogeneous contract graphs 118, 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. The heterogeneous control-flow graphs 30 and call graphs 20 used in the present approach can be derived from the abstract syntax trees (ASTs) 112, for example generated by tree-sitter or similar programs which parse the source code or byte code of smart contracts 10, which is received via a user interface, an API, or other input software module. Thus, the generated heterogeneous contract graphs 118 do not depend on specific compilers because they only rely on the grammar and syntax of programming languages. In addition, graph-generating and graph-learning components used in the present approach remain independent of the languages and can be applied to new types of vulnerabilities.

[0026] Prior approaches employed, for example, Slither static analyzer tool to generate control-flow graphs 30 and call graphs 20. The approach requires the compilation of smart contracts, which occasionally precipitate version conflicts and the absence of necessary libraries, needing more human effort to address compilation-related issues. To address this limitation, the present approach uses a tree-sitter instead, a syntax-based graph generation tool to establish heterogeneous control-flow graphs 30 and heterogeneous call graphs 20 from ASTs. However other programs for generating abstract syntax trees 112 from source code of byte code may be used, including ANTLR, Roslyn (C#/VB), Solc AST (Solidity), Babelfish, neovim, etc.

[0027] LLMs are 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. The present approach relies on two components in an LLM: its 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, thereby facilitating bug identification and comprehension.

[0028] When existing deep and graph neural network techniques learn bug patterns from specific code representations, such as syntax trees and data/control dependency graphs, they typically treat trees/graphs as flattened sequences or disjointing conventional graphs, 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. In contrast, the present approach accurately captures vulnerability patterns and locate fine-grained vulnerabilities at the line level. In addition, the prior art approach of using predefined patterns may only detect specific bugs, such as Reentrancy and Time Manipulation. In contrast, our frameworks employ a heterogeneous graph structure, enabling a more general and flexible exploration of various vulnerability types without needing predefined patterns.

[0029] Figure 2A show five components for the architecture used by an electronic platform in accordance with one embodiment of the present approach: the Heterogeneous Contract Graph Generator 110, the Meta Relations Extractor 120, the Node Features Extractor 130, the Graph Neural Network 140, and the Two-Phase Vulnerability Detector 150. The input consists of the source code 10 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. The five components are described in detail below.

[0030] Heterogeneous Contract Graph Generator 110.

[0031] The first component in Figure 2A, Heterogeneous Contract Graph Generator 110, is responsible for processing the received source code or byte code of a smart contract to output a representation of the smart contract code using a heterogeneous contract graph 118 based on the fusion of heterogeneous Control-Flow Graphs (CFGs) 30 and heterogeneous Call Graphs (CGs) 20. The present 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. A heterogeneous graph is defined as follows:

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 $|R| \geq 1$.

[0032] The generation of heterogeneous contract graph (HCG) 118 is initiated based on an Abstract Syntax Tree (AST) 112 produced by a parser generator tool and an incremental parsing library. From generated ASTs, the heterogeneous contract graph generator 110 collects a pre-defined set of node types 22 and establishes edges that follow the control flow of the source code to create Control Flow Graphs (CFGs) 30.

Similarly, this component can extract the nodes that are invoked to each other via the function call edges to create Call Graphs (CGs) 20. The control-flow graphs and call graphs are provided or translated into a heterogeneous representation 114 which uses sets of node types 22 and edge types 24 to preserve and retain most of the structure and semantics of the source code. These are then fused together by sub-component Heterogeneous Graphs Fusion 116. This fused graph is called a heterogeneous contract graph (HCG) 118. The nodes thus represent a semantic or syntactical segment of the smart contract code and each edge represents a control flow from a first node to a second node.

[0033] Heterogeneous Control-Flow Graphs 30.

[0034] A heterogeneous control-flow graph (CFG) 30 may encompass various types of statements or lines of code. We employ typical statement types as node types 22 within the source code's heterogeneous control-flow graphs, such as *EXPRESSION*, *NEW_VARIABLE*, *RETURN*, *IF*, *END_IF*, *IF_LOOP*, and *END_LOOP* or other similar logical tests or loop related control statements. A set of edge types 24 are used to indicate connections between nodes that describe the sequential or branching nature of statements, including *NEXT*, *TRUE*, and *FALSE*. An illustrative example of a generated heterogeneous control-flow graph 30 for the Collect function within the MY_BANK contract 10 is presented in Figure 1B. This comprises five node types 22, namely *FUNCTION_NAME* 22a, *NEW_VARIABLE* 22d, *EXPRESSION* 22e, *IF* 22f, and *END_IF* 22g, and three edge types 24, namely *TRUE* 24a, *FALSE* 24b and *NEXT* 24c. Due to the capabilities and limitations of parser generator tools, an heterogeneous control-flow graph for a source code is generated on a per-function basis for a contract. To aggregate the heterogeneous control-flow graphs for all functions within a contract, we incorporate call graphs 20 for source code, as further elaborated below.

[0035] Heterogeneous Call Graphs 20.

[0036] A heterogeneous call graph (CG) 20 represents the invocation relations among functions within one or multiple smart contracts. The present approach framework acknowledges two primary forms of calls in the context of smart contracts: *internal_calls* 24e, which encompass function calls within the same contract, and *external_calls* 24f, which pertain to function calls that traverse across different contracts. These two call types are represented by two distinct edge types, namely *INTERNAL_CALL* 24e and *EXTERNAL_CALL* 24f. To distinguish between the functions of a smart contract and external functions, we introduce a node called *CONTRACT_DECLARATION* 22b at the top of each contract. This node is linked to all its function definitions through *define_method* edges 24d. In addition to the typical function node type *FUNCTION_NAME* 22a, we also incorporate the *FALLBACK_NODE* 22c 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. Figure 1B illustrates an example of such a

heterogeneous call graph 20 for the *MY_BANK* contract. This comprises three node types 22, namely *CONTRACT_DECLARATION* 22b, *FUNCTION_NAME* 22a and *FALLBACK* 22c and three edge types 24, namely *DEFINE_METHOD* 24d, *INTERNAL_CALL* 24e and *EXTERNAL_CALL* 24f.

[0037] Heterogeneous Contract Graphs (HCGs) 118.

[0038] 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 the present approach, the sub-component Heterogeneous Graphs Fusion 116 combines the heterogeneous call graphs 20 and heterogeneous control-flow graphs 30 by establishing a connection (linking edge) between the *FUNCTION_NAME* nodes 22a within both heterogeneous call graphs 20 and heterogeneous control-flow graphs 30 that represent the same function (as demonstrated by the double-arrow edge 22h linking the Collect node in the heterogeneous call graph 20 and the Collect Node in the heterogeneous control-flow graph 30 in Figure 1B. That is for each and every function node 22a in a call graph 20, a linking edge 22h is created to the corresponding function node 22a at the entry node of the control-flow graph, and thus the heterogeneous call graph 20 is expanded with the heterogeneous control-flow graph 30 to include the granular control flow details of the heterogeneous control-flow graph 30, resulting in the generation of a heterogeneous contract graph (HCG) 118. The set of node types and set of edge types in the heterogeneous contract graph (HCG) 118 is thus a union of the types found in the heterogeneous control-flow graphs 30 and heterogeneous call graphs 20.

[0039] Meta Relations Extractor 120

[0040] The Meta Relations Extractor component is responsible for extracting customized meta relations from the generated heterogeneous contract graphs (HCGs) 118. A meta relation is defined as:

A meta relation of an edge $e = (s, t)$ from a source node s to a target node t is indicated as $\langle \pi(s), \phi(e), \pi(t) \rangle$, with $\pi(s)$ and $\pi(t)$ 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.

[0041] The meta relations are dynamically determined based on node types and edge types present in the heterogeneous contract graph, and thus do not need to rely on predefined relation schemas providing enhanced flexibility (i.e. are dynamically determined without relying on a predefined relation schema). The main advantage of extracting these meta relations 141 is to mitigate the potential explosion of all possible node type 22 and edge type 24 combinations, a common issue in the traditional approaches employing metapath. This is particularly crucial since the number of node types 22 and edge types 24 in the graphs is dynamic and in this embodiment can encompass up to eighteen node types and five edge types (these numbers could be varied in other embodiments). The meta relations 141 are provided as inputs to the HGT layer 144 to generate the embeddings.

[0042] 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 1B can be described by both $\langle \text{EXPRESSION}, \text{next}, \text{END_IF} \rangle$ and $\langle \text{EXPRESSION}, \text{back}, \text{END_IF} \rangle$. HCGs 118 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) 144 employed in the present approach. This is because the original architecture of each HGT layer requires at least two source nodes for one target node. Without reflective relations, many nodes with only one source node (e.g., the two *EXPRESSION* nodes in Figure 1B) would be overlooked during training. The meta relations extractor 120 thus processes the heterogeneous contract graph and outputs (or defines) how connected nodes interact, for example by defining the sets of nodes and edges that can be grouped together. The extracted meta relations are then provided as explicit inputs to the graph neural network 140 and directly influence relation-aware attention weights and message propagation between heterogeneous node types.

[0043] Node Features Extractor 130

[0044] The primary objective of this extractor 130 is to generate initial features 143 for each node, enriching their properties within the graphs. In this module, we leverage the capabilities of large language models trained on vast corpora, specifically the tokenizer of LLMs trained on corpora (training dataset) including one or more programming languages (e.g. solidity, vyper, yul, rust, python, Javascript, C++, etc).

[0045] 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 132 of the LLMs to encode and extract the node attribute features 134 from the code snippets inside nodes rather than employing whole heavy 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 and analyze smart contracts without high computational cost-effectively. As outlined above, the tokenizer parses the source code and splits the text into tokens according to the predetermined vocabulary for the tokenizer, and each text token is mapped to a numeric token ID. We then look up the associated embedding vector for each token found in the source code (the node attribute embeddings 134) from the embedding matrix or embedding layer of the pretrained LLM. For a given pre-trained LLM, the embedding vector is a fixed length (e.g. 768, 1024, 2048, etc.) and stores the embeddings of the tokens learned during the pre-training of the LLM (stored as the embedding matrix, or as a layer in the LLM). The embedding vector encodes or captures the semantic meaning of the token (i.e., relationships with other tokens in the vocabulary). Similar tokens will have similar vector representations. The embedding vector is thus a numerical representation

of the token in a high dimensional space capturing semantic meaning, and in a full LLM is passed into the multiple layers of the LLM model. We pass the node features 143 (the embedding vector for each token in the source code) as inputs to the graph neural network 140.

[0046] Each LLMs' tokenizer has its own predetermined vocabulary with a fixed size (and which can vary from LLM to LLM), which was pre-trained (determined) by a special technique. Byte-Pair Encoding (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 a desired vocabulary size is reached or the tokenizer no longer finds frequent pairs of bytes to merge, thus creating the predetermined vocabulary of the LLM. 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. In the examples described below we selected three LLMs as the baselines shown in Tables 2 and 3 and their respective tokenizer based on three variants of BPEs as Node Feature Extractor that have certain benefits.

[0047] Solidity-t5: The variant of the T5 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 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.

[0048] StarCoder: A powerful LLM boasting 15.5 billion parameters pre-trained on 80+ programming languages, excluded solidity whose tokenizer is an original BPE 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.

[0049] CodeT5p-770m: This model is a part of the CodeT5 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.

[0050] Solidity-t5 was chosen 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.

[0051] The specific pretrained language models described herein are exemplary, and other tokenizers or embedding models trained on programming language corpora may be employed.

[0052] On the other hand, fine-tuning the LLMs in a new language for a new task requires many computing resources. To address the limitations, the present approach leverages 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.

[0053] Graph Neural Network 140

[0054] In this embodiment the graph-based machine learning model is a Graph Neural Network (GNN) and is based on a Heterogeneous Graph Transformer (HGT) model (algorithm). We input all pairs of meta relations 141 for each target node, including their node types and node features (e.g. embedding vector) 143, into a single HGT layer 144. This approach enables the GNN to learn the interrelations among our customized meta relations in HCGs. This disentangles complex node/edge relationships, ultimately aiding in detecting actual bugs in a smart contract. The outputs from the GNN are then passed on to the final component, the Two-Phase Vulnerability Detector, to identify whether the smart contract contains bugs and to estimate the locations of the bug(s) within the contract source code.

[0055] Heterogeneous Graph Transformer (HGT) layer 144.

[0056] 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 . To achieve this, the architecture of Transformer [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] 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. 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^h GNN layer, denoted as $H^{(l-1)}[t]$, $H^{(l-1)}[s_1]$ and $H^{(l-1)}[s_2]$. A more detailed model of the HGT layer is shown in Figure 2B. There are three sub-components within the HGT layer:

[0057] Heterogeneous Mutual Attention 145: 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.

[0058] Heterogeneous Message Passing 146: 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.

[0059] Target-Specific Heterogeneous Message Aggregation (Target Specific Aggregation) 147: 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 148 to generate the embedding vector $H^{(L)}[t]$ 148'. Such a mechanism ensures that the final node embedding vector for node type t considers multiple aspects through the Transformer architecture. We thus obtain node embeddings for each node type 149

[0060] Optimization for Detection 149': We use a multi-layer perceptron with the softmax function as an activation function for graph or node classification tasks (on the node embeddings for each node type 149). 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.

[0061] Two-Phase Vulnerability Detector 150

[0062] This component comprises two primary phases (detector software modules): Coarse-Grained Detection 152 and Fine-Grained Detection 154 and returns a detection result (output) indicating if the presence of one or more bugs were detected in the source codes, and estimates (or determines) a location of the respective bug in the source code. The Coarse-Grained Detection phase assesses clean versus vulnerable smart contracts at the contract level. This may output a binary classification such as clean (i.e. correctly written) or buggy (having one or more bugs). The classification may be hard classification or a soft classification such as a probability of at least one bug being present. In some embodiments the classifier may be a multiclass classifier, such as clean, buggy or unclassifiable. The unclassifiable class may indicate the smart contract code vectors were unrecognisable (or indeterminate) against the corpus of the correct smart contract code and the buggy smart contract code. The Fine-Grained Detection phase 154 estimates the precise line locations of vulnerabilities in the contract source code, so that bugs can be identified up to the line level, in comparison to earlier learning-based methods which only report vulnerabilities at the contract or function level.

[0063] Phase 1: Coarse-Grained Detection 152.

[0064] In this phase, the goal is to determine whether a smart contract is vulnerable (i.e. has at least one bug). We employ the heterogeneous contract graphs and their corresponding embeddings for each input smart contract. We train an MLP 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 a first threshold, for example 0.5. Other thresholds could be used, and may be set by the user (e.g. via a user interface or input parameter) depending upon the desired level of confidence in identifying a bug, and may be higher or lower than 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.

[0065] Phase 2: Fine-Grained Detection 154.

[0066] 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/estimated/predicted as buggy when its confidence score with respect to a specific bug type exceeds a second threshold, for example 0.5. Importantly, the nodes correspond to statements or lines in the source code, allowing us to estimate/identify the locations of vulnerabilities at the line level within the source code precisely. Other second thresholds could be used and may be set by the user (e.g. via a user interface or input parameter) depending upon the desired level of confidence in identifying a bug, and may be higher or lower than 0.5. Additionally, the first and second thresholds need not be the same. For example, the first threshold could be selected to allow low confidence estimates (i.e., potentially a high false positive rate) whilst the second threshold could be a higher confidence. This may increase the number of false positives, but increase the user's confidence in reviewing the code. Alternatively, the first threshold could be set to be high confidence value to allow the user to focus on high confidence bugs. The user interface or API may allow the user to iteratively vary the first and second thresholds to control the number of bugs that need to be reviewed. For example, a user could first look for high confidence estimates, and if there is available review time, then go searching for lower confidence bug estimates. Once a bug is identified, the user can then review and update the code to correct the bug, or if they determine the estimate is wrong (no bug) they can move to the next estimate.

[0067] Evaluation

[0068] An embodiment of smart contract vulnerability detector 100 implementing the model architecture shown in Figures 2A and 2B was generated and executed on an electronic platform and the performance was evaluated on two datasets comprising different mixtures of the SmartBugs Curated, the SolidiFI-Benchmark, the Clean Smart Contracts from SmartBugs, and the DAppScan datasets. Details on these datasets as follows:

[0069] SmartBugs Curated [Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts in ACM/IEEE 42nd International Conference on Software Engineering. 530–541; and João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: a framework to analyze solidity smart contracts, in 35th IEEE/ACM International Conference on Automated Software Engineering 1349–1352.]: a collection of vulnerable Ethereum smart contracts organized into nine types. It contains 143 annotated contracts with 208 tagged vulnerabilities.

[0070] SolidiFI-Benchmark [Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection, in 29th ACM SIGSOFT International Symposium on Software Testing and Analysis]: 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.

[0071] Clean Smart Contracts from SmartBugs Wild [Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts in ACM/IEEE 42nd International Conference on Software Engineering. 530–541; and João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: a framework to analyze solidity smart contracts, in 35th IEEE/ACM International Conference on Automated Software Engineering 1349–1352.]: We identified 2,742 contracts out of the 47,398 Ethereum smart contracts that do not contain any bugs based on eleven SmartBugs integrated detection tools. Thus, we used the 2,742 contracts as a set of clean contracts.

[0072] DAppScan 2024 dataset [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>]. The DAppScan dataset is a large-scale dataset with 39,904 Solidity files from 682 real-world DApp projects, featuring 1,618 SWC weaknesses. 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. [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>]. After mapping we have 614 buggy solidity files.

[0073] Additional testing was also performed on the GPTScan 2024 [Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis." In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1-13. 2024. <https://github.com/GPTScan/GPTScan>] and DeFiHacksLab 2025 [SunWeb3Sec, “Reproduce DeFi hack incidents using Foundry,” <https://github.com/SunWeb3Sec/DeFiHackLabs>, Last updated in Dec 2025].

[0074] To ensure consistency in the evaluation, we focus on the seven types of vulnerabilities that are joint in the datasets, including Access Control, Arithmetic, Denial of Service, Front Running, Reentrancy, Time Manipulation, and Unchecked Low-Level Calls. In other embodiments other bug types may be detected. These seven bug types are briefly described below:

[0075] 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.

[0076] 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.

[0077] Arithmetic - 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.

[0078] 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.

[0079] 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.

[0080] 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.

[0081] 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.

[0082] All smart contracts within the datasets were provided in source code form. Using a parser generator tool called tree-sitter and an incremental parsing library, we parsed each source file into Abstract Syntax Trees (ASTs). The heterogeneous contract graph generator 110 was further configured to perform the following tasks: Generate Control Flow Graphs 30 and Call Graphs 20 for Solidity smart contracts as well as a fusion form combining such two graph types before extracting meta relations and feeding them to the GNN component.

[0083] Additionally, we introduced a random mixing strategy where we selected some smart contracts from the clean dataset and combine them with the buggy dataset. Specifically, we selected 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 resulted in two composite datasets: Dataset A, which includes buggy contracts from both SolidiFI-Benchmark and SmartBugs Curated, and Dataset B, which includes buggy contracts from SolidiFI-Benchmark and DAppSCAN. We maintain a balanced ratio 1:1 between clean and buggy contracts, ensuring that our training and test datasets are more evenly balanced for both graph and node classification tasks. Table 1 shows an overview of the actual number of buggy contracts and total number of contracts used in our experiments for each bug type for Dataset A, as well as

the total numbers of nodes and edges in the constructed heterogeneous contract graphs for source code. Table 2 shows a similar overview for Dataset B.

TABLE 1

Statistics of the mixed dataset A (SmartBugs Curated, the SolidiFI-Benchmark, and the Clean Smart Contracts from SmartBugs datasets)

Bug Types	# Total / Buggy Contracts	# Total Nodes	# Total 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 2

Statistics of the mixed dataset B (clean smart contracts from the SmartBugs Wild, the SolidiFI-Benchmark, and the DAppSCAN datasets)

Bug Types	# Total / Buggy Contracts	# Total Nodes	# Total 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

[0084] Our approach requires line-level labels for the bugs as opposed to datasets that only provide coarse-grained contract- or function-level labels for the bugs.

[0085] Evaluation Metrics

[0086] 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:

$$F1 = \frac{2 \times precision \times recall}{precision + recall},$$

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

[0087] 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.

[0088] Baselines and Parameter Settings

[0089] To demonstrate the advantages of heterogeneous graph learning over conventional Graph Neural Networks (GNNs) and Large Language Models (LLMs), we selected three conventional GNNs models from GAE [Thomas N Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. NIPS Workshop on Bayesian Deep Learning (2016)], LINE [Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding], and node2vec [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], 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 [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](https://doi.org/10.1016/0364-0213(90)90002-E)], LSTM [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>], and GRU [Kyunghyun Cho, Bart van Merriënboer, 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>], 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 used the encoders of three LLMs: Solidity-t5, StarCoder, and CodeT5p-770m to classify buggy code at both coarse-grained and fine-grained levels. We employed the pretrained model encoders to embed source code snippets, and then we fine-tuned the last hidden state with classification layers. Finally, we calculated the average of the individual smart contracts to represent the source file. Additionally, we selected the best-performing result from 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, as our method of comparison. We also compared our line-level source code bug detection method to six widely used smart contract vulnerability detection tools based on conventional software analysis techniques: *Manticore* [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], Mythril [Bernhard Mueller. 2018. Smashing Smart Contracts for Fun and Real Profit. In 9th annual HITB Security Conference. 2–51.], Oyente [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], Securify [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], Slither [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], and SmartCheck [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]. 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 began “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”. The bug categories were listed and briefly explained with an example, and the model was further instructed that if a vulnerability exists, the model was to clearly indicate where in the code and why (in 1-2 sentences), and to focus on clarity correctness and reasoning.

[0090] Parameter Settings.

[0091] All models in our experiments use an input node feature size of 512 to keep more information of the code inside a node from LLM 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 incorporated 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 GNN of Figure 2A, we applied two HGT layers and configured eight multi-heads. 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 were 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.

[0092] Experimental Results

[0093] 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).

[0094] Coarse-Grained Contract-Level Vulnerability Detection.

[0095] Tables 3 and 4 displays the Buggy-F1 scores and Macro F1 scores of the baselines and the electronic platform of Figure 2A at the contract level on Dataset A and B.

TABLE 3

Performance comparison in terms of Buggy-F1 score (first line) and Macro-F1 (second line) score on different bug detection methods at the contract granularity level on dataset A. We use the Heterogeneous Contract Graphs of both clean and buggy smart contracts as the inputs.

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	31.64	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
LLMs only	solidity-t5	35.00	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
Present approach with Node Features Generated by LLM tokenizers	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 4

Performance comparison in terms of Buggy-F1 score (first line) and Macro-F1 (second line) score on different bug detection methods at the contract granularity level on dataset B. We use the Heterogeneous Contract Graphs of both clean and buggy smart contracts as the inputs.

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional GNNs	GCN	0.00 33.33	38.2 48.73	65.24 32.62	56.6 28.3	62.93 31.46	0.00 30.66	0.00 31.48
	LINE	0.00 33.92	66.26 34.85	0.00 34.93	64.07 44.28	74.35 37.17	0.00 32.46	62.96 31.48
	node2vec	59.4 36.08	62.96 64.24	61.31 30.65	68.96 34.48	71.89 35.94	0.00 29.72	69.02 34.51
Conventional RNNs	Vanilla-RNN	63.64 67.00	32.43 46.16	76.11 78.48	62.92 66.86	65.55 68.44	31.82 49.95	67.37 68.64
	LSTM	0.00 32.41	68.66 51.85	64.17 32.09	40.00 54.09	0.00 31.41	0.00 30.3	64.38 32.19
	GRU	66.67 61.57	43.64 55.33	0.00 30.6	67.35 68.29	51.49 60.53	0.00 32.65	0.00 32.65
LLMs only	solidity-t5	0.00 35.08	56.41 28.20	63.30 31.65	63.06 31.53	70.19 35.09	0.00 35.49	49.35 47.61
	starcoder	5.12 35.59	71.67 37.79	68.05 34.02	70.08 35.04	67.56 33.78	0.00 34.17	63.63 67.18
	codet5p-770m	0.00 32.11	60.00 30.00	62.31 31.15	67.82 33.91	65.75 32.87	0.00 36.58	42.69 28.12
Prompting LLMs	DeepSeek	54.50 74.40	44.40 70.70	52.60 75.50	44.40 71.60	58.80 76.80	0.00 50.00	0.00 47.60
	GPT-4.0	54.50 74.30	47.00 71.09	15.00 57.10	25.00 62.50	66.70 83.30	0.00 50.00	0.00 44.00
	Gemini Pro	30.80 61.10	23.50 57.40	0.00 47.70	15.40 51.10	50.00 50.00	0.00 50.00	74.10 69.60
The best Buggy F1 scores of MANDO-HGT	Node features of the best scores	83.63 81.63	80.99 81.14	87.88 87.49	88.46 87.98	88.52 89.33	87.97 87.74	82.35 81.99
Present approach with Node Features	solidity-t5	92.30 91.80	88.89 89.18	90.91 90.47	90.20 90.00	89.55 89.22	94.12 94.28	88.37 89.64
	starcoder	86.79 85.61	89.47 89.33	90.91 90.62	90.57 90.18	88.89 89.37	94.12 94.28	83.33 83.67

Generated by LLM tokenizers	codet5p-770m	85.71 85.71	84.93 85.32	94.12 93.73	88.00 88.23	92.54 92.30	97.14 97.06	88.46 87.98
-----------------------------	--------------	----------------	----------------	------------------------------	----------------	------------------------------	------------------------------	-----------------------

[0096] Since each source file typically contains one smart contract, we treat each source file as a smart contract. We observe that an embodiment of the present approach, using node features generated by the LLMs’ tokenizers, outperforms the conventional GNNs, conventional RNNs, and LLMs on both datasets. It achieves higher scores than MANDO-HGT. For instance, the present approach improves Buggy-F1/Macro-F1 scores in the time manipulation bug by 35.52%/38.75 and 97.14%/64.6% compared to the best score of three conventional GNNs, by 31.77%/58.31% and 65.32%/47.11% compared to the best score of 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, and by 3.81%/4% and 9.17%/9/32% compared to the best score of MANDO-HGT. On the Dataset B, we outperform all three industrial LLMs: DeepSeek, GPT-4.0, and Gemini Pro on all seven types of vulnerabilities. Moreover, regardless of which tokenizer is used to generate node features, the present approach consistently outperforms the baselines. We suppose that the performance of the present approach does not significantly suffer from the type of tokenizers generated by LLMs.

[0097] It is evident that integrating node features from different tokenizers inside the electronic platform of Figure 2A 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.

[0098] Fine-Grained Line-Level Vulnerability Detection.

[0099] Tables 5 and 6 shows the performances of the present approach and other baselines at the line level for Datasets A and B.

TABLE 5

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

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional Detection Tools	security	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	50	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	80
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 F1 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 only	solidity-t5	85.01	86.14	84.78	84.08	85.21	83.29	64.11
	starcode	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
Present approach with Node Features Generated by LLM tokenizers	solidity-t5	95.39	95.42	98.23	94.69	95.41	97.29	96.18
	Starcode	96.49	93.96	98.11	95.66	94.26	96.17	94.97
	codethp-770m	94.53	94.33	8.19	94.95	96.78	96.40	96.62

TABLE 6

Performance comparison in terms of Buggy-F1 scores across various bug detection methods based on source code at the line granularity level of the Dataset B. The highest score for each bug type is in bold.

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional Detection Tools	securily	16.23 23.01	13.71 21.85	14.62 22.83	9.35 21.32	14.48 22.65	8.42 19.28	9.81 20.14
	mythril	13.46 22.11	12.23 21.28	10.17 18.96	9.58 20.51	15.91 23.37	7.74 18.14	11.12 19.7
	slither	18.17 24.76	14.82 22.64	15.45 23.89	12.61 22.16	17.34 24.91	10.97 21.29	11.68 23.1
	manticore	10.84 21.06	11.15 21.48	11.93 22.7	8.37 20.39	10.61 22.13	7.59 18.75	8.84 19.62
	smartcheck	11.72 22.35	10.31 21.42	12.36 23.45	9.89 21.7	9.57 20.86	8.96 20.16	8.68 20.39
	oyente	9.87 20.23	9.93 20.41	10.91 21.75	8.61 19.34	11.09 22.98	8.17 18.09	7.95 18.75
Conventional GNNs	GCN	11.16 45.82	10.1 46.33	31.88 17.85	0.19 41.72	7.47 42.61	0 40.19	27.39 15.13
	LINE	3.34 43.85	30.77 41.9	11.84 49.14	47.9 49.07	40.26 42.48	18.23 46.93	27.16 19.89
	node2vec	38.38 44.58	28.43 53.16	25.01 51.32	54.96 53.7	33.32 53.09	36.14 47.64	32.64 49.14
LMM Encoder	solidity-t5	4.75 44.9	3.47 46.07	0.27 44.98	41.51 57	30.6 50.86	2.93 41.2	2.71 47.01
	starcoder	3.99 44.25	21.73 52.16	0 44.76	14.11 43.24	8.2 45.65	1.93 41	1.72 46.55
	codet5p-770m	5.05 44.82	2.96 45.88	0.08 44.7	18.87 45.88	0.06 41.62	33.26 54.21	0.33 45.6
Prompting LLMs	DeepSeek	40.82 61.11	31.37 63.9	41.36 69.09	36.86 65.07	52.36 70.25	20.11 53.56	13.66 53.13
	GPT-4.0	33.33 56.45	24.43 24.43	4.98 40.52	21.24 57.86	51.9 69.98	8.17 44.45	13.04 51.58
	Gemini Pro	29.41 53.94	15.87 56.15	2.59 37.56	16.55 52.87	40.07 62.31	6.67 35.34	5.42 41.55

The best Buggy F1 scores of MANDO-HGT	Node features of the best scores	59.59 76.73	77.67 84.27	61.68 76.79	80.4 83.44	77.2 83.84	76.43 82.79	90.67 94.53
Present approach with Node Features Generated by LLM tokenizers	solidity-t5	75.29 86.13	88.21 91.44	71.31 82.29	92.79 94.4	85.47 89.6	82.38 87.11	87.93 92.85
	Starcoder	71.69 84.02	88.47 91.71	67.35 79.9	92.82 94.43	83.4 88.02	82.26 87.15	91.14 94.8
	codethp-770m	71.78 84.08	87.65 91.15	72.4 83.08	91.73 93.54	84.77 89.12	82.54 87.27	89.32 93.72

[0100] From the results, we observe that:

- The present approach generally outperforms conventional analysis-based bug detection tools, basic GNNs and LLMs encoder-only baselines on both Datasets A and B. 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. 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. Our approach also performs good results 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, the present approach still performs consistently and outperforms the baselines.

[0101] Interpreting Vulnerability Prediction Results

[0102] We examined 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).

[0103] Average attention scores

[0104] We utilized 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 the present approach 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 Figure 1A 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 analyzed some meta relations with a significant difference in average attention score between clean and buggy smart contracts.

[0105] Figure 3 is a plot 300 of the 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. The slanted columns represent the attention scores for the clean smart contracts and the filled columns represent the attention scores for the reentrancy smart contracts, and the dashed line and solid line represent the number of edges for the clean and reentrancy smart contracts respectively. 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 (dark slanted and black filled) 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 1A containing many edges with considerable gaps.

[0106] The $\langle FUNCTION_DEFINITION, next, NEW_VARIABLE \rangle$ relation has a 0.15 average 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 code snippets are shown in Table 7 below, while Figure 5 shows a node graph corresponding to the code snippet of etherbank.sol (the boxed portion). Figure 5 also shows node scores of the etherbank.sol smart contract.

TABLE 7

Code snippet of the smart contracts containing a reentrancy vulnerability

Code snippet of smart contract 0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888.sol
<pre> 55 function Collect(uint _am) public payable 56 { 57 var acc = Acc[msg.sender]; 58 if(acc.balance>=MinSum && acc.balance>=_am && now>acc.unlockTime) 59 { 60 // <yes> <report> REENTRANCY 61 if(msg.sender.call.value(_am)()) 62 { 63 acc.balance-=_am; 64 LogFile.AddMessage(msg.sender,_am,"Collect"); 65 } 66 } 67 } </pre>
Code snippet of etherbank.sol
<pre> 18 function withdrawBalance() { 19 unit amountToWithdraw = userBalances[msg.sender]; 20 // <yes> <report> REENTRANCY 21 if(!(msg.sender.call.value(amountToWithdraw)())) { throw;} 22 userBalances[msg.sender] = 0; 23 } </pre>
Code snippet of reentrancy_simple.sol
<pre> 27 function withdrawBalance() { 28 // send userBalance[msg.sender] ethers to msg.sender 29 // if mgs.sender is a contract, it will call its fallback function 30 // <yes> <report> REENTRANCY 31 if(! (msg.sender.call.value(userBalance[msg.sender])())){ 32 throw; 33 } 34 userBalance[msg.sender] = 0; 35 } </pre>

[0107] 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.

[0108] Additionally, we observed the differences in attention score gap between two specific vulnerability types. Figure 4 is a plot 400 showing the attention score gaps inside reentrancy and access control buggy smart contracts. The number of edges correspond to these meta relations. The meta relations that have under 100 edges in both types were filtered. Reentrancy bugs are shown in solid black columns and access control bugs are shown in columns filled with slanted lines, along with the number of edges corresponding to these meta relations where the reentrancy edge amount is shown by the dashed line and the access control edge amount shown by the solid black line. 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 dotted box in Figure 4), but are absent in access control smart contracts. In contrast, the meta relations $\langle FUNCTION_DEFINITION, next, EXPRESSION \rangle$, $\langle NEW_VARIABLE, next, EXPRESSION \rangle$, $\langle IF, pos_next, EXPRESSION \rangle$, $\langle IF, pos_next, RETURN \rangle$, $\langle CONTRACT_DECLARATION, next, NEW_VARIABLE \rangle$, and $\langle NEW_VARIABLE, next, RETURN \rangle$ (meta relations inside the long dash dot dot box in Figure 4) 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, the present approach is a vulnerability-specific approach.

[0109] We employed GStarX [Shichang Zhang, Neil Shah, Yozen Liu, and Yizhou Sun. 2022. Explaining Graph-level Predictions with Communication Structure-Aware Cooperative Games. arXiv:2201.12380 [cs.CL]], a Structure-Aware explainer for GNNs, to improve the explanation of the present approach 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. Figure 5 shows a node graph 500 with the nodes 502 and node scores for the `etherbank.sol` smart contract. The graph part inside in the dashed box 506 corresponds to the code snippet shown in Table 7. The dashed border nodes 504 in Figure 5 were considered the most important nodes with positive scores from GStarX when the present approach detects 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.

[0110] In summary, embodiments as described herein provides 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. Using the tree-sitter parser generator to establish heterogeneous control-flow graphs and call graphs from ASTs, the present approach possesses the inherent capability to accommodate new programming languages through a set of grammatical rules. The present approach simultaneously exploited the structured code awareness of the HCGs being the combinations of control-flow graphs and call graphs 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, leveraging 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.

[0111] It will also be understood that the embodiment described above, and as illustrated in Figure 2A may be varied. For example, alternative neural network architectures and machine learning models can be used, which are trained to use the graph-based outputs of the meta relations extractor 120 and tokenizer based node feature extractor 130 on a training dataset comprising examples of buggy and correct (or clean) smart code contracts. The alternative neural network architectures and machine learning models include graph-based architectures and models (i.e. specifically designed to handle graph-based data), or general architectures and models adapted/reconfigured for processing graph-based data or which pre-process graph data into format usable by such architectures and models, or hybrid systems that combine graph-based architectures and models (or components) with general architectures and models (or components). For example a range of Artificial Intelligence (AI) and Machine Learning (ML) models may be used, and configured to use a range of techniques such as classification, pattern recognition, clustering, etc, and include the use of random forests, K-nearest neighbour (KNN), decision trees, support vector machine, Naïve Bayes, as well as neural network and AI models including graph neural networks (GNN), Heterogeneous Graph Transformers (HGT), graph convolutional networks (GCN), graph attention networks (GAT), Message passing neural networks (MPNNs), convolutional neural networks (CNN), recurrent neural networks (RNN), feedforward neural networks (FNN), transformers, autoencoders, and generative adversarial network (GAN). These models may each be trained to identify vulnerabilities in smart code contracts implement the architecture illustrated in Figure 2A and as described above.

[0112] Those of skill in the art would understand that information and signals may be represented using any of a variety of technologies and techniques. For example, data, instructions, commands, information, signals, bits, symbols, and chips may be referenced throughout the above description may be represented by voltages, currents, electromagnetic waves, magnetic fields or particles, optical fields or particles, or any combination thereof.

[0113] A model may be trained using a standard AI/Machine Learning training approach. An input dataset is generated which contains examples of buggy and clean/correct smart contract code. The input dataset may undergo a pre-pre-processing step to convert it to suitable input form, for example a heterogeneous graph representation including meta relations and node features. Other preprocessing includes scaling, normalisation, augmentation, etc may be performed. The input dataset then undergoes a test/train split step in which a training dataset is generated and used to training the ML model, and a test dataset is held back and used for testing the trained model. In some embodiments a blind validation dataset may be split from the input dataset and used only to test the final trained ML model. Typical test/train splits are largely an arbitrary choice, although typically more data is used to train the ML model than used to test a trained model and splits such as 60:40, 70:30 and 80:20 is relatively common. If a blind validation dataset is used, it is typically smaller than the test dataset and thus test/train/validate splits may be 60:30:10 or 70:20:10.

[0114] A model training process is then conducted for multiple epochs using the training dataset to obtain a trained ML model which estimates an outcome measure from the input variables, for example if the smart contract has a vulnerability, and where in the smart contract the vulnerability is. An initial model architecture is selected, and node weights and hyper parameters are selected. For example, if the ML model is a neural network model the architecture may comprise an initial flattening layer, followed by n hidden dense layers each comprised of multiple neurons, and an output layer. An activation function such as a rectified linear unit (ReLU) may be employed between neurons in adjacent layers. Hyperparameters such as a learning rate, a loss function may be selected for use during the training process. The number of epochs is typically chosen to ensure the ML model has been sufficiently exposed to the available data for training and generalisation purposes. The trained ML model then undergoes model evaluation where the accuracy of the trained model is tested using the test dataset, in which the trained model is provided with the input variables and estimated outcome measure is compared to the known outcome measure. A hyperparameter adjustment process is then performed using an optimiser which adjusts the architecture and hyperparameters such as the learning rate, number of neurons within a dense layer, number of dense layers, activation function, and number of epochs. The loop of model training on the training dataset, followed by model evaluation and hyperparameter adjustment is repeated until a threshold number of epochs is reached or an accuracy threshold condition is met (e.g. change in accuracy is less than a threshold). The trained ML model is then output. As noted above, if a blind validation dataset was retained, the accuracy of the final

trained ML model may be assessed using the blind validation dataset. Variations on this approach may also be used.

[0115] Those of skill in the art would further appreciate that the various illustrative logical blocks, modules, circuits, and steps described in connection with the embodiments disclosed herein may be implemented as electronic hardware, computer software or instructions, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and steps have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present disclosure.

[0116] The steps of a method described in connection with the embodiments disclosed herein may be embodied directly in hardware, in a software module executed by a processor, or in a combination of the two. For a hardware implementation, processing may be implemented within one or more application specific integrated circuits (ASICs), digital signal processors (DSPs), digital signal processing devices (DSPDs), programmable logic devices (PLDs), field programmable gate arrays (FPGAs), processors, controllers, micro-controllers, microprocessors, or other electronic units designed to perform the functions described herein, or a combination thereof.

[0117] The electronic platform 160 may be a computer implemented system and may comprise one or more processors 162 including multi-core CPUs and Graphical Processing Units (GPUs) operatively connected to one or more memories 164 which store instructions to configure the processor to perform embodiments of the method. In this context, the electronic platform (or computing system) may comprise one or more computer apparatus including, for example, one or more processors (CPUs, GPUs), memories, storage, and input/output devices (e.g., monitor, keyboard, disk drive, network interface, Internet connection, etc.) which may be a unitary device or distributed apparatus. A CPU may comprise an Input/Output Interface, an Arithmetic and Logic Unit (ALU) and a Control Unit and Program Counter element which is in communication with input and output devices through the Input/Output Interface. The Input/Output Interface may comprise a network interface and/or communications module for communicating with an equivalent communications module in another device using a predefined communications protocol (e.g. Bluetooth, Zigbee, IEEE 802.15, IEEE 802.11, TCP/IP, UDP, etc.). The computing apparatus may comprise a single CPU (core) or multiple CPU's (multiple core), or multiple processors. The computing apparatus may be a cloud based computing apparatus using GPU clusters, a parallel processor, a vector processor, or be a distributed computing device. Memory may be operatively coupled to the processor(s) and may comprise RAM and ROM components and may be provided within or external to the device or processor module. The memory may be used to store an operating system and

additional software modules or instructions. The processor(s) may be configured to load and executed the software modules or instructions stored in the memory. A computer program (or code) may be written, for example, in a general-purpose programming language, e.g., Python, R, Julia, Scala, Java, Javascript, Ruby, C++, etc., or some specialized application-specific language, and may utilise or call software libraries or packages, including NumPy, Pandas, Matplotlib, Scikit-learn, Keras, PyTorch, TensorFlow, Transforms, Cuda, NLTK, Langchain, Theano, SonarQube, Checkmarx, ESLint, CodeClimate, GitHub Copilot, Snyk Code, Codiga, etc.

[0118] The electronic platform may be a computing apparatus such as an all-in-one computer, desktop computer, laptop, tablet or mobile computing apparatus, server, and any associated peripheral devices. The computer system may be a distributed system including computing apparatus configured to capture and process data from server-based systems and cloud-based computing systems. A distributed system or apparatus may comprise several components operatively (or functionally) connected via wired or wireless connections and include cloud-based systems. In some operational settings, the computing system may be configured as a system that includes one or more devices, each of which is configured to carry out some aspects of the processes either in software, hardware, or some combination thereof. A user interface may be provided on a desktop computer, tablet computer or mobile computing apparatus, and be configured to receive source code and to display results or perform further analysis, or be used to configure, train and update AI/ML models. The user interface may be provided as a web portal and be configured to allowing users to submit source code and to provide an output interface identifying any bugs/vulnerabilities in the source code, i.e. a detection result from the graph neural network indicating if one or more bugs were detected in the source codes, and a location of the respective bug in the source code. Similarly an application programming interface (API) may be provided to receive source code from a user or a program and to provide the output (detection result) identifying any bugs/vulnerabilities in the received source code.

[0119] Software modules, also known as computer programs, computer codes, or instructions, may contain a number of source code or object code segments or instructions, and may reside in any computer readable medium such as a RAM memory, flash memory, ROM memory, EPROM memory, registers, hard disk, a removable disk, a CD-ROM, a DVD-ROM, a Blu-ray disc, or any other form of computer readable medium. In some respects, the computer-readable media may comprise non-transitory computer-readable media (e.g., tangible media). In addition, for other aspects computer-readable media may comprise transitory computer-readable media (e.g., a signal). Combinations of the above should also be included within the scope of computer-readable media. In another aspect, the computer readable medium may be integral to the processor. The processor and the computer readable medium may reside in an ASIC or related device. The software codes may be stored in a memory unit, and the processor may be configured to execute them. The memory unit may be implemented within the processor or external to the processor, in which case it can be communicatively coupled to the processor via various means as is known in the art.

[0120] The memory may be used to store executable software code, as well as operating system and additional software modules or instructions. The processor(s) may be configured to load and executed the software modules or instructions stored in the memory.

[0121] Further, it should be appreciated that modules and/or other appropriate means for performing the methods and techniques described herein can be downloaded and/or otherwise obtained by a computing device. For example, such a device can be coupled to a server to facilitate the transfer of means for performing the methods described herein. Alternatively, various methods described herein can be provided via storage means (e.g., RAM, ROM, a physical storage medium such as a Flash Drive, optical disc (DVD, CD) or etc.), such that a computing device can obtain the various methods upon coupling or providing the storage means to the device. Moreover, any other suitable technique for providing the methods and techniques described herein to a device can be utilized.

[0122] The methods disclosed herein comprise one or more steps or actions for achieving the described method. The method steps and/or actions may be interchanged with one another without departing from the scope of the claims. In other words, unless a specific order of steps or actions is specified, the order and/or use of specific steps and/or actions may be modified without departing from the scope of the claims.

[0123] As used herein, the terms “estimating” or “determining” encompasses a wide variety of actions. For example, “estimating” or “determining” may include calculating, computing, processing, deriving, investigating, looking up (e.g., looking up in a table, a database or another data structure), ascertaining and the like. Also, “estimating” or “determining” may include receiving (e.g., receiving information), accessing (e.g., accessing data in a memory) and the like. Also, “determining” may include resolving, selecting, choosing, establishing and the like.

[0124] In the application, unless specified otherwise, the terms "comprising", "comprise", and grammatical variants thereof, intended to represent "open" or "inclusive" language such that they include recited elements but also permit inclusion of additional, non-explicitly recited elements.

[0125] The reference to any prior art in this specification is not, and should not be taken as, an acknowledgement or any form of suggestion that such prior art forms part of the common general knowledge.

[0126] In some cases, a single embodiment may, for succinctness and/or to assist in understanding the scope of the disclosure, combine multiple features. It is to be understood that in such a case, these multiple features may be provided separately (in separate embodiments), or in any other suitable combination. Alternatively, where separate features are described in separate embodiments, these separate features may be combined into a single embodiment unless otherwise stated or implied. This also applies to the claims which can be recombined in any combination. That is a claim may be amended to include a feature defined in any other claim. Further a phrase referring to “at least one of” a list of items refers to any combination

of those items, including single members. As an example, “at least one of: a, b, or c” is intended to cover: a, b, c, a-b, a-c, b-c, and a-b-c.

[0127] It will be appreciated by those skilled in the art that the disclosure is not restricted in its use to the particular application or applications described. Neither is the present disclosure restricted in its preferred embodiment with regard to the particular elements and/or features described or depicted herein. It will be appreciated that the disclosure is not limited to the embodiment or embodiments disclosed, but is capable of numerous rearrangements, modifications and substitutions without departing from the scope as set forth and defined by the following claims.

CLAIMS

1. An electronic platform for detecting bugs in smart contract code, the electronic platform comprising:
 - at least one memory; and
 - at least one processor configured to execute instructions stored in the memory that configure the at least one processor to:
 - process received smart contract code to generate a heterogeneous contract graph comprising a plurality of nodes and at least one edge, where each node represents a semantic or syntactical segment of the smart contract code and each edge represents a control flow from a first node to a second node, wherein the plurality of nodes comprises at least two node types, and the at least one edge comprises at least one edge type;
 - generate a plurality of node attribute features for each node by processing each node with a tokenizer of a pretrained Large Language Model (LLM), wherein the pretrained LLM is a LLM trained on a dataset comprising source code of at least one programming language and comprises a predetermined vocabulary and an embedding matrix or layer, and processing each node with the tokenizer comprises generating one or more tokens for each node using the predetermined vocabulary of the pretrained LLM and returning a vector of the embeddings for each of the one or more tokens using the embedding matrix or layer;
 - extract a plurality of meta relations from the nodes, the edges or both defining how connected nodes interact wherein the meta relations are dynamically determined based on node types and edge types present in the heterogeneous contract graph;
 - process the plurality of meta relations and the vectors with a trained graph-based machine learning model, which was trained on a training dataset comprising meta relations and vectors of correct smart contract code and buggy smart contract code, and the trained graph-based machine learning model is configured to perform heterogeneous message passing and relation-aware aggregation across node types and edge types and to detect a presence of one or more bugs in the received smart contract code and for each bug estimate a location of the respective bug in the source code; and
 - return a detection result indicating if one or more bugs were detected in the source codes, and a location of the respective bug in the source code.
2. The electronic platform as claimed in claim 1, wherein processing the received smart contract code comprises parsing the smart contract code and generating an abstract syntax tree representation of the smart contract code, wherein the received smart contract code is received in source code or byte code forms.

3. The electronic platform as claimed in claim 1 or 2, wherein generating the heterogeneous contract graph comprises generating one or more heterogeneous control-flow graphs and one or more heterogeneous call graphs and fusing the one or more heterogeneous control-flow graphs and the one or more heterogeneous call graphs to form the heterogeneous contract graph, wherein each heterogeneous control-flow graph represents one or more functions, statements or lines of code, and each heterogeneous call graph represents an invocation relation between a first function and either a second function within the smart contract or with a second function in an external smart contract, where each node has a node type selected from a set of at least two node types, and each edge has an edge type selected from a set of at least one edge type, and the fusing comprises forming an edge between a node in a first heterogeneous control-flow graph with a node in a first heterogeneous call graph that represents the same function as the node in the first heterogeneous control-flow graph.
4. The electronic platform as claimed in any one of claims 1 to 3, wherein the graph-based machine learning model is based on a graph neural network.
5. The electronic platform as claimed in claim 4, wherein the graph neural network is based on a heterogeneous graph transformer.
6. The electronic platform as claimed in any one of claims 1 to 5, wherein detecting a presence of one or more bugs in the received smart contract code is performed in two stages where a first stage detects if at least one bug is present in the smart contract code and a second stage determines a location in the source code of each of the at least one bugs.
7. The electronic platform as claimed in any one of claims 1 to 6, wherein the one or more bugs comprise any one or more of the following bugs: Access Control, Arithmetic, Denial of Service, Front Running, Reentrancy, Time Manipulation, and Unchecked Low-Level Calls.
8. The electronic platform as claimed in any one of claims 1 to 7, wherein the meta relations further define sets of nodes and edges that are grouped together.
9. The electronic platform as claimed in claim 8, wherein the grouped nodes and edges comprise adjacent nodes with reflective connections.

10. The electronic platform as claimed in any one of claims 1 to 9, wherein the at least two node types includes any one or more of the following: a function; a new variable, an expression; a logical test, a loop related statement; a fallback or a contract declaration, and the at least one edge type includes any one or more of the following: Next, Back, True, False, a define method, an internal call, an external call, a linking edge.

11. A computational method for detecting bugs in smart contract code, the method comprising:
 processing received smart contract code to generate a heterogeneous contract graph comprising a plurality of nodes and at least one edge, where each node represents a semantic or syntactical segment of the smart contract code and each edge represents a control flow from a first node to a second node, wherein the plurality of nodes comprises at least two node types, and the at least one edge comprises at least one edge type;

generate a plurality of node attribute features for each node by processing each node with a tokenizer of a pretrained Large Language Model (LLM), wherein the pretrained LLM is a LLM trained on a dataset comprising source code of at least one programming language and comprises a predetermined vocabulary and an embedding matrix or layer, and processing each node with the tokenizer comprises generating one or more tokens for each node using the predetermined vocabulary of the pretrained LLM and returning a vector of the embeddings for each of the one or more tokens using the embedding matrix or layer;

extracting a plurality of meta relations from the nodes, the edges or both defining how connected nodes interact wherein the meta relations are dynamically determined based on node types and edge types present in the heterogeneous contract graph;

processing the plurality of meta relations and the vectors with a trained graph-based machine learning model, which was trained on a training dataset comprising meta relations and vectors of correct smart contract code and buggy smart contract code, and the trained machine learning model is configured to perform heterogeneous message passing and relation-aware aggregation across node types and edge types and to detect a presence of one or more bugs in the received smart contract code and for each bug estimate a location of the respective bug in the source code; and

returning a detection result indicating if one or more bugs were detected in the source codes, and a location of the respective bug in the source code.

12. The method as claimed in claim 11, wherein processing the received smart contract code comprises parsing the smart contract code and generating an abstract syntax tree representation of the smart contract code, wherein the received smart contract code is received in source code or byte code forms.

13. The method as claimed in claim 11 or 12, wherein generating the heterogeneous contract graph comprises generating one or more heterogeneous control-flow graphs and one or more heterogeneous call graphs and fusing the one or more heterogeneous control-flow graphs and the one or more heterogeneous call graphs to form the heterogeneous contract graph, wherein each heterogeneous control-flow graph represents one or more functions, statements or lines of code, and each heterogeneous call graph represents an invocation relation between a first function and either a second function within the smart contract or with a second function in an external smart contract, where each node has a node type selected from a set of at least two node types, and each edge has an edge type selected from a set of at least one edge type, and the fusing comprises forming an edge between a node in a first heterogeneous control-flow graph with a node in a first heterogeneous call graph that represents the same function as the node in the first heterogeneous control-flow graph.
14. The method as claimed in claim 11, 12 or 13, wherein the graph-based machine learning model is based on a graph neural network.
15. The method as claimed in claim 14, wherein the graph neural network is based on a heterogeneous graph transformer.
16. The method as claimed in any one of claims 11 to 15, wherein detecting a presence of one or more bugs in the received smart contract code is performed in two stages where a first stage detects if at least one bug is present in the smart contract code and a second stage determines a location in the source code of each of the at least one bugs.
17. The method as claimed in any one of claims 11 to 16, wherein the one or more bugs comprise any one or more of the following bugs: Access Control, Arithmetic, Denial of Service, Front Running, Reentrancy, Time Manipulation, and Unchecked Low-Level Calls.
18. The method as claimed in any one of claims 11 to 17, wherein the meta relations further define sets of nodes and edges that are grouped together.
19. The method as claimed in claim 18, wherein the grouped nodes and edges comprise adjacent nodes with reflective connections.
20. The method as claimed in any one of claims 11 to 19, wherein the at least two node types includes any two or more of the following: a function; a new variable, an expression; a logical test, a loop related

statement; a fallback or a contract declaration, and the at least one edge type includes any one or more of the following: Next, Back, True, False, a define method, an internal call, an external call, a linking edge.

21. A computer readable medium comprising instructions for configuring at least one processor to perform the method of any one of claims 11 to 20.