Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

SmartScope: Smart contract vulnerability detection via heterogeneous graph embedding with local semantic enhancement

Zhaoyi Meng, Zexin Zhang, Wansen Wang*, Jie Cui, Hong Zhong

School of Computer Science and Technology, Anhui University, Anhui, China

A R T I C L E I N F O

Keywords:
Solidity smart contracts
Vulnerability detection
Heterogeneous graph
Representation learning
Semantic enhancement

A B S T R A C T

Smart contracts are integral to blockchain ecosystems, yet their security remains a critical concern due to the prevalence of exploitable vulnerabilities. Existing conventional and deep learning-based vulnerability detection methods often struggle to capture the fine-grained semantics and heterogeneous structural dependencies essential for accurate analysis. We propose and implement *SmartScope*, a novel technique for smart contract vulnerability detection that leverages heterogeneous graph embedding with local semantic enhancement. Specifically, SmartScope constructs a semantically rich contract graph that depicts control-flow, data-flow, and fallback relations among critical code elements. To guide the graph learning process, we empirically assign various importance coefficients to vulnerability-relevant subgraphs, thereby enhancing the detection model's focus on semantically critical regions. The heterogeneous graph transformer is then employed to generate context-aware node representations, which are then passed to an MLP-based detector for vulnerability classification. To the best of our knowledge, this is the first method that structurally encodes domain knowledge into the heterogeneous graph learning for achieving effective smart contract analysis. Experimental results demonstrate that SmartScope outperforms 10 representative conventional and deep learning-based baselines on over 5K smart contracts. The evaluation spans multiple vulnerability types, including reentrancy, timestamp dependence, and infinite loops, highlighting the effectiveness and robustness of our work.

1. Introduction

Blockchain technology has transformed digital transactions beyond its initial cryptocurrency applications. While Bitcoin (Vranken, 2017) introduces limited programmable capabilities, the development of Turing-complete smart contracts marks a significant advancement in blockchain functionality. These self-executing programs have proliferated rapidly, with Ethereum alone hosting over 78 million deployed contracts as of June 2025, collectively managing assets worth thousands of billions of dollars (Etherscan, 2025). Smart contracts have extended blockchain applications across finance, commerce, healthcare, logistics, and legal domains, enabling unprecedented levels of automation, transparency, and disintermediation in business processes (Lin et al., 2022).

Smart contracts have become prime targets for attackers due to their control over valuable cryptocurrency assets (Ivanov et al., 2023). This unique security challenge is fundamentally driven by three factors: (1) the enormous financial incentive for exploitation, (2) the permission-less nature of blockchain networks, where contract code is publicly accessible to anyone, and (3) the immutability of blockchain, which makes vulnerable contracts difficult to patch once deployed. For instance, in April 2025, KiloEx decentralized exchange lost approximately 7.5 million USD in a cross-chain attack due to a smart contract price oracle vulnerability that allowed price manipulation (Blockonomi, 2025). In March 2025, a decentralized exchange aggregator named 1inch suffered an attack due to a reentrancy vulnerability labeled by SolidityScan (Shashank, 2025) in its resolver smart contracts based on the outdated Fusion v1 implementation, leading to an approximate loss of 5 million USD (Langley, 2025). These incidents indicate the urgent need for proactively identifying vulnerabilities before deployment to ensure the security of smart contracts.

Various conventional methods for detecting smart contract vulnerabilities have been proposed, *e.g.*, formal verification (Liu et al., 2025; Stephens et al., 2021; Wang et al., 2023), static analysis (Liao et al., 2024, 2022; Wang & Zhao, 2025), fuzzing (Li et al., 2024a; Torres et al., 2021; Ye et al., 2024), and symbolic execution (He et al., 2022; Tsankov et al., 2018; Zheng et al., 2022). However, these methods primarily rely on expert-defined patterns to identify potential issues. While these approaches prove to be effective in specific scenarios, they suffer from

* Corresponding author.
E-mail addresses: zymeng@ahu.edu.cn (Z. Meng), E23301201@stu.ahu.edu.cn (Z. Zhang), 23762@ahu.edu.cn (W. Wang), cuijie@mail.ustc.edu.cn (J. Cui), zhongh@ahu.edu.cn (H. Zhong).

https://doi.org/10.1016/j.eswa.2025.129857
Received 5 August 2025; Received in revised form 12 September 2025; Accepted 22 September 2025
Available online 27 September 2025

fundamental limitations: manually defined patterns may be error-prone and struggle to handle unseen cases (Zhuang et al., 2021). As a result, sophisticated attackers can easily circumvent these detection mechanisms using various tricks. More critically, as the number of smart contracts in blockchain ecosystems grows exponentially, it becomes impractical to rely solely on expert knowledge to thoroughly analyze contracts and define precise detection rules, and then incorporate them into analysis tools.

Recent years have witnessed growing interest in deep learning-based methods for detecting smart contract vulnerabilities precisely. Early attempts utilized LSTM networks to analyze contracts sequentially (Luo et al., 2022; Qian et al., 2020), which cannot preserve critical structural information of contract code. Although an LLM-based method (Ding et al., 2025) advances smart contract vulnerability detection, it fails to explicitly model structural information and struggles to capture long-range semantic dependencies within contract code. Another category of methods build specified graphs based on smart contract code and apply standard graph neural networks for the detection (Lakadawala et al., 2024; Liu et al., 2021b; Wang et al., 2024b; Zhuang et al., 2021). These methods cannot effectively capture the heterogeneity among different types of nodes and edges, leading to the loss of key semantic information. To address the problem, numerous heterogeneous graph-based smart contract vulnerability detection techniques have been proposed (Li et al., 2024b; Luo et al., 2024; Nguyen et al., 2023; Zhu et al., 2025). However, the built heterogeneous graphs lack sufficient granularity, making it difficult to extract essential semantics for vulnerability detection. For example, SCVHunter does not perform fine-grained modeling of conditional statements in smart contract code, which can lead to missed detections of timestamp dependency vulnerabilities involving the use of *block.timestamp* in branch conditions. To further improve detection performance, some methods (Liu et al., 2021a,b) separately encode expert knowledge and graph structural information into vectors and fuse them via concatenation. However, such feature-level fusion is hard to fully exploit domain knowledge to guide the graph representation learning.

We propose *SMARTSCOPE*, a novel method for smart contract vulnerability detection via heterogeneous graph embedding with local semantic enhancement. Our goal is to explicitly enhance the semantic representations of vulnerability-related subgraphs by assigning them greater weights based on domain knowledge, thereby guiding the detection model to better capture critical features for accurate identification. While the overall design is conceptually intuitive, it significantly helps the model learn more meaningful and vulnerability-aware representations. To the best of our knowledge, this is the first method that structurally encodes domain knowledge into the heterogeneous graph learning for achieving effective smart contract analysis.

To implement SMARTSCOPE, we firstly build a semantically rich contract graph (SRCG) by integrating multiple types of semantic relations among program statements, including data dependencies, control dependencies, function calls, and temporal execution sequences. Nodes in the graph represent critical program elements, *e.g.*, function invocations, variables, or branch points, while edges capture their temporal and semantic interactions. Compared with the coarse-grained graph structures (*e.g.*, aggressive node merging) constructed in previous studies, SRCG decomposes smart contract code into semantically distinct nodes (*e.g.*, control nodes for conditional statements) and further connects them via multiple types of edges (*e.g.*, control-flow edges for function calls). This fine-grained and heterogeneous graph structure enables SRCG to preserve both structural and semantic relations within smart contracts that are often abstracted away in prior coarse-grained approaches, thereby providing a more expressive and precise foundation for vulnerability detection. We further propose a prior-guided embedding enhancement strategy that leverages domain knowledge about the causes of specific vulnerabilities to promote graph representation learning. This is achieved by empirically amplifying the initial embeddings of semantically important nodes in the SRCG. Unlike existing methods that are purely data-driven or directly concatenate expert knowledge with graph representations at the feature level, our approach structurally integrates domain knowledge into the graph learning process, resulting in more informative and vulnerability-aware representations. The enhanced embeddings are then fed into the downstream heterogeneous graph transformer (HGT) model for highlighting suspicious subgraphs. Finally, a multi-layer perceptron (MLP) is employed to identify vulnerabilities based on the learned graph embeddings.

As a proof of concept, we use SMARTSCOPE to detect reentrancy, timestamp dependence, and infinite loop vulnerabilities. These three types are representative of distinct and prevalent vulnerability categories in real-world smart contracts (Liu et al., 2021b; Luo et al., 2024). We evaluate SMARTSCOPE on a reasonably sized dataset of 5073 smart contracts whose labels are manually verified by domain experts. The contracts are collected from 3 open-source datasets (Liu et al., 2021b; Luo et al., 2024; Qian et al., 2023a), along with a part of synthetic samples generated via LLMs (Chen et al., 2025) to enhance coverage. Extensive experiments are conducted, comparing SMARTSCOPE with 5 state-of-the-art tools based on traditional techniques and 5 deep learning-based approaches. The results demonstrate that SMARTSCOPE achieves detection accuracy rates of 96.60 %, 96.91 %, and 98.48 % for the mentioned vulnerabilities, respectively, outperforming the selected baselines significantly. Additionally, we validate the effectiveness of building the SRCG and assigning different importance coefficients through ablation studies.

Our main contributions are summarized as follows:

- We propose and implement SMARTSCOPE, a novel technique for smart contract vulnerability detection that leverages the heterogeneous graph embedding with local semantic enhancement.
- To the best of our knowledge, this is the first work to structurally encode domain knowledge into the heterogeneous graph learning process for achieving effective smart contract analysis.
- Our comprehensive evaluations on more than 5K smart contract samples with 10 baselines demonstrate the effectiveness of SMARTSCOPE.

The paper is structured as follows: Section 2 presents the problem statement of our work. Section 3 details the methodology. Section 4 reports our experimental results. Section 5 discusses our limitations. Section 6 shows the related work, and Section 7 concludes.

2. Problem statement

Given the source code of a smart contract, we are interested in developing an automatic approach to determine whether the contract contains potential vulnerabilities. Specifically, we are to estimate the label $\hat{y}$ for each smartuy contract SC, where $\hat{y} = 1$ represents SC has a specific vulnerability, while $\hat{y} = 0$ indicates SC is safe. We focus on three types of vulnerabilities, including reentrancy, timestamp dependence, and infinite loop. Although these vulnerabilities have been widely studied in previous work (Liu et al., 2021b; Luo et al., 2024; Wang et al., 2024a,c; Zhuang et al., 2021), they continue to pose significant security threats in modern smart contract systems (Fox, 2024). The subtle semantic characteristics of vulnerabilities pose non-trivial challenges to existing detection approaches. Furthermore, this work targets Solidity smart contracts (Dannen, 2017), the dominant language in the Ethereum, which has been linked to numerous real-world security issues. We next present detailed descriptions of the three types of vulnerabilities.

(1) *Reentrancy* is a well-known vulnerability in smart contracts, where a function performs an external call (*e.g.*, *call.value()*) to another contract before updating its internal state. If the callee implements a fallback function, it can re-enter the caller's function during the same execution context, potentially causing repeated invocations and inconsistent state updates or unauthorized fund transfers.

(2) *Timestamp Dependence* arises when a smart contract relies on the block timestamp as a condition to trigger critical operations, *e.g.*,

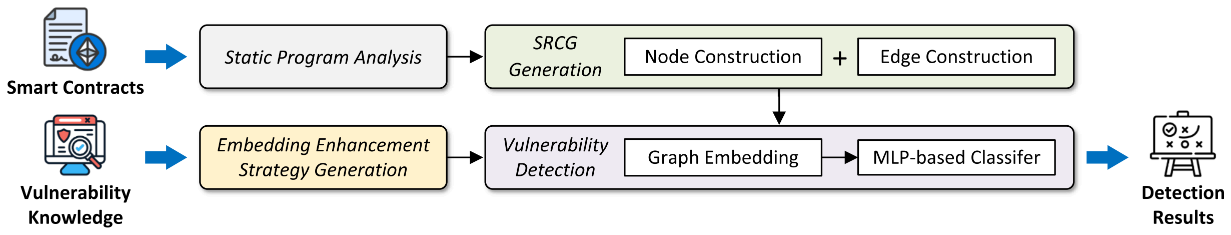


Fig. 1. Overall architecture of SMARTSCOPE.

transferring Ether or determining the winner of a lottery ticket. Since the miner who produces the block can manipulate the timestamp within a permissible range (typically less than 900 seconds) (Jiang et al., 2018), this flexibility may be exploited to gain unfair or unauthorized advantages.

(3) *Infinite Loop* is a typical vulnerability in smart contracts, where a loop (*e.g.*, *for()*, *while()*, or self-invocation loop) lacks a valid termination condition. On Ethereum, such loops may consume excessive gas, especially when containing costly operations like *CALL* (Chen et al., 2021), potentially leading to denial-of-service or failed transactions due to gas limit exhaustion.

3. Methodology

3.1. Architecture

As shown in Fig. 1, the overall architecture of SMARTSCOPE consists of four main components as follows:

- *Static program analysis* extracts control-flow, data-flow, and fallback-associated relations from Solidity source code of smart contracts based on the Slither framework (Feist et al., 2019). The extracted information is essential for constructing SRCG. As this component mainly relies on existing techniques, we will not elaborate on its details in the following.
- *SRCG generation* generates nodes and edges based on the information extracted before. To distinguish semantic roles of different elements in smart contracts, we design 4 types of nodes and 3 types of edges. In addition, a temporal number is assigned to each edge to precisely reflect execution orders and dependence relations among the nodes (Zhuang et al., 2021). The generated SRCG with the fine-grained semantics above facilitates the identification of various types of vulnerabilities in smart contracts.
- *Embedding enhancement strategy generation* produces a prior-guided embedding enhancement strategy that promotes the downstream graph representation learning based on domain knowledge about the causes of specific vulnerabilities. Specifically, we firstly extract vulnerability-prone subgraphs from the SRCG based on prior knowledge of common vulnerability patterns. We then empirically assign different importance coefficients to enhance the semantic expressiveness of nodes within these subgraphs. The enhancement strategy is finally fed to the next component for guiding the graph representation learning.
- *Vulnerability detection* firstly adjusts node embeddings within the subgraphs by scaling their initial representations with the precomputed coefficients. Subsequently, the HGT model is employed to capture the heterogeneity of the SRCG. Fine-grained modeling of contract code, combined with the enhancement of critical node embeddings, enables the resulting representations to capture rich semantic information. These embeddings are then used to train a standard MLP-based classifier to identify vulnerabilities in smart contracts. As demonstrated in the proposed works (Liu et al., 2021a; Luo et al., 2024), MLP is typical and effective to deal with high-dimensional data in this context.

3.2. SRCG generation

3.2.1. Motivation

While prior work has introduced various graph construction methods for smart contracts, their coarse-grained modeling hinders the accuracy of downstream vulnerability detection tasks. For example, Zhuang et al. (2021) and Liu et al. (2021b) model smart contract code as contract graphs. To highlight critical nodes, they eliminate non-critical nodes and combine the involved features, resulting in a simplified structure. However, the removed contextual information is often vital for reasoning about vulnerabilities (Nguyen & Vo, 2024). Non-critical nodes can provide complementary semantics, and the aggressive elimination may compromise detection effectiveness. Although Nguyen et al. (2022a,b, 2023) utilize heterogeneous graph neural networks for vulnerability detection, their approach of converting each line of code as a node leads to the loss of fine-grained semantics (*e.g.*, specific operations of variables) that are crucial for accurate detection. Luo et al. (2024) explicitly model data dependency relations, but omit the construction of the nodes for certain conditional statements (*e.g.*, *if()*, *assert()*, and *require()*), which also affect the detection accuracy.

To overcome these limitations, we design a fine-grained heterogeneous graph named SRCG tailored for smart contract analysis. Specifically, we define 4 types of nodes that comprehensively cover various critical code elements, based on the representations of code semantics and the requirements of vulnerability detection. We then introduce 3 types of edges to precisely capture the diverse interaction relations between the code elements within the nodes. We also incorporate temporal information into the edge construction to preserve the sequential characteristics of different types of flows according to the execution order of the code. To sum up, we comprehensively maintain code information relevant to vulnerability detection in the SRCG. The heterogeneous graph structure enables us to distinguish the roles and contributions of different types of nodes and edges, thereby facilitating the downstream detection task.

3.2.2. Node construction

As explained before, different program elements in smart contracts contribute unequally to vulnerability detection. Therefore, to highlight semantic differences among code elements, we extract 4 types of nodes representing multiple distinct element categories, as listed in Table 1.

Basic Nodes. Basic nodes represent various types of variables that are important for detecting specific vulnerabilities. For example, to detect timestamp dependence vulnerability, the global variable *block.timestamp* deserves more attention than the other local variables. To capture the semantic differences among variables, we model each variable as a basic node with its actual content. In comparison, previous studies often eliminate variable nodes (Zhuang et al., 2021) or treat all variables as the same node type without detailed content (Luo et al., 2024), thereby overlooking their contextual distinctions.

Action nodes. Given that invocations of customized or build-in functions are central to the execution of smart contracts, action nodes are designed to capture the core intentions of contract code and then facilitate the detection of hidden vulnerabilities. For instance, analyzing reentrancy vulnerabilities often requires focusing on Ether transfer

Table 1
Details of our constructed nodes.

Node Type	Element
Basic Nodes	Global, local, and parameter variables
Action Nodes	Functions of *send()* and *transfer()* for fixed-gas Ether transfers Functions of *call.value()* for unrestricted Ether transfers User-defined functions in smart contracts
Control Nodes	Entry points of smart contract function execution Statements of *if()* in smart contract code Exit points of conditional blocks Statements of *assert()* and *require()* for enforcing conditions Entry points of loop structures End points of loop structures Loop conditions Statements of *continue* Statements of *break* Statements of *return*
Fallback Nodes	Fallback functions

Table 2
Summarization of semantic edges, where `X` denotes a placeholder for logical expressions, conditions, or variables involved in the corresponding semantic construct.

Edge Type	Semantic Fact

Control-flow Edges	`assert{X}`
	`require{X}`
	`revert`
	`throw`
	`if{X}`
	`if{...}else{X}`
	`if{...}then{X}`
	`while{X}do{...}`
	`for{X}do{...}`
	Natural sequential relationships

Data-flow Edges	`assign{X}`
	`access{X}`
Fallback Edges	Interactions with fallback function

mechanisms, including high-level methods like *transfer()* and low-level calls like *call.value()* (Cai et al., 2025). We therefore customize a corresponding action node for each function call. Prior approaches adopt coarse-grained schemes of the functions in smart contracts (*e.g.*, only using core nodes for various function calls without the invoking objects Liu et al., 2021b), which may lead to the loss of critical behavior semantics.

Control nodes. Control nodes are used to depict conditional branches in smart contract code, which often serve as important indicators for identifying vulnerabilities. For example, modeling the loop structure is essential for identifying infinite loop vulnerabilities, which often manifest through unbounded control flows. The lack of explicit modeling of control nodes in existing work results in a loss of branching semantics (Liu et al., 2021b; Luo et al., 2024; Zhuang et al., 2021), which compromises the effectiveness of the vulnerability detection. Hence, we model a control node for each branch statement.

Fallback nodes. Many smart contract vulnerabilities (*e.g.*, reentrancy) are directly or indirectly related to the fallback function. However, these vulnerabilities often surface only during interactions with malicious contracts. Based on existing work (Liu et al., 2021a,b; Luo et al., 2024), we build a fallback node to simulate the fallback function that inspires a virtual attack contract and interacts with the function under test.

3.2.3. Edge construction

Nodes are not isolated, instead, they are tightly connected through various relations in a temporal order. To capture rich semantic dependencies between the aforementioned nodes, we design 3 types of edges, including *control-flow edge*, *data-flow edge*, and *fallback edge*. Each edge represents a potential execution path within the function under analysis, and its temporal index reflects the position of the edge in the function's execution sequence. We analyze various function structures and summarize the semantics of edge construction in Table 2, which is similar to the prior work (Liu et al., 2021b; Luo et al., 2024).

Control-flow edges. The edges encapsulate control semantics of smart contract code, particularly suited for conditional and security-check statements (*e.g.*, *if()*, *for()*, *assert()*, *require()*). A control-flow edge points from the preceding node, representing a critical function call or variable, to the one in the current statement, thereby capturing critical control logic in the execution order of contract code. In this category, we also adopt forward edges to model the natural control flows in the code for preserving the logical structure expressed in the original code sequence. A forward edge connects two nodes in the adjacent statements, either within the same function (*e.g.*, sequential statements), or across functions by linking a call site to the entry point of the callee function.

Data-flow edges. The edges trace variable usage and changes, including both accesses and modifications. Data-flow edges model the propagation of data throughout the program and reveal dependencies between variables.

Fallback edges. To model the fallback mechanism specific to Solidity, we introduce two dedicated fallback edges. The first edge links the first occurrence of *call.value()* to the corresponding fallback node. The second edge connects the fallback node to the caller function that contains the *call.value()* invocation. This modeling helps reveal potential fallback logic paths that can be exploited by attackers.

3.2.4. Node and edge features

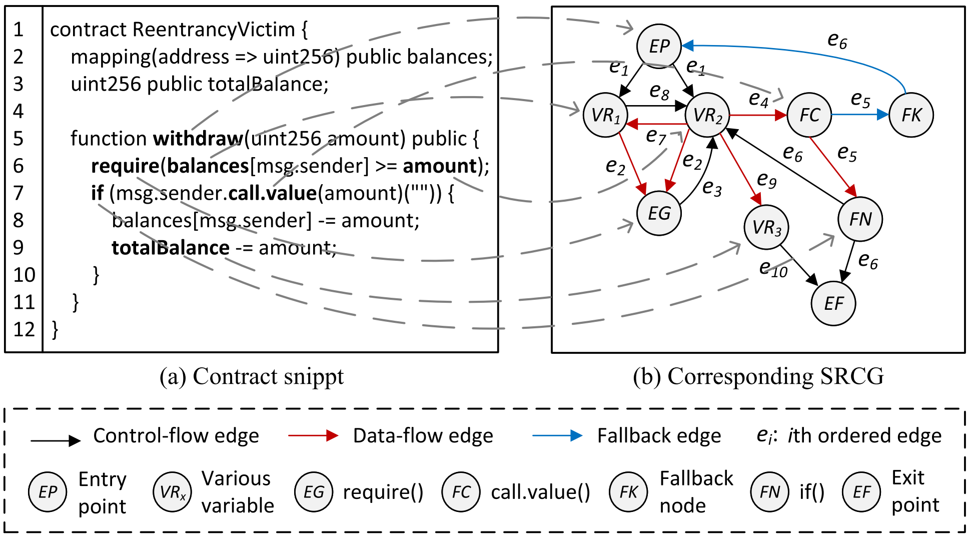
The SRCG construction process extracts different features based on the previously described node and edge definitions. Specifically, the feature of a node is shown as a tuple (ID, t, e), where ID denotes its identifier, t stands for the node type, and e presents the element of the code within the node. In contrast to prior approaches (Liu et al., 2021b; Luo et al., 2024) that extract different features for different node types, we propose a unified representation that facilitates systematic modeling and detection of smart contract vulnerabilities. The feature of an edge is described as a tuple (V_s, V_e, o, t), where V_s and V_e indicate its start and end nodes, o denotes its temporal order, and t represents its edge type.

3.2.5. A representative example

To illustrate the details of the SRCG, we use Fig. 2 as a representative example. Fig. 2(a) shows a smart contract snippet that enables Ether withdrawals based on user balances. However, the snippet transfers funds via call before updating internal state, introducing a reentrancy vulnerability. An attacker can exploit this by recursively invoking withdraw to drain the contract's balance.

To depict the semantics of the code snippet precisely, we plot the corresponding SRCG as shown in Fig. 2(b). For clarity, gray dashed arrows are used to highlight the mapping between code elements and the corresponding nodes in the SRCG. The symbol on each node represent its detailed type, while the arrows in different colors represent control-flow, data-flow, and fallback edges respectively. Note that the elements of the nodes is not drawn due to space constraints. The label on an edge (*i.e.*, e_i) indicates its sequential order (*i.e.*, ith) during the function execution. Note that it is possible for two edges to share the same label in the SRCG. For example, for the two edges of $FN \xrightarrow{e_6} VR_2$ and $FN \xrightarrow{e_6} EF$, the former depicts the control-flow behavior triggered when the conditional expression at Line 7 of code evaluates to true, while the latter illustrates the behavior when the expression is false. As the result of the conditional expression is uncertain in static analysis, we apply identical labels to both branches to capture both execution semantics. Similarly, the edges of $VR_1 \xrightarrow{e_2} EG$ and $VR_2 \xrightarrow{e_2} EG$ have the same label.

Overall, the SRCG comprehensively models the key semantic information in smart contract code while preserving the semantic



```
contract ReentrancyVictim {
   mapping(address => uint256) public balances;
   uint256 public totalBalance;

   function withdraw(uint256 amount) public {
      require(balances[msg.sender] >= amount);
      if (msg.sender.call.value(amount)("")) {
         balances[msg.sender] -= amount;
         totalBalance -= amount;
      }
   }
}
```

Fig. 2. SRCG construction of our method based on real-world smart contract code.

independence of each node, which is essential for further vulnerability detection. In contrast, the aforementioned approaches tend to remove or merge certain nodes, potentially leading to the loss of critical contextual information. Although the SRCG adopts a fine-grained modeling strategy to preserve rich semantic information, the overall information volume remains limited due to the relatively small size of smart contract code. Moreover, we will adopt some embedding enhancement strategies for improving the representation of semantically meaningful local subgraphs. As a result, such models can still be effectively processed by advanced deep neural networks.

3.3. Embedding enhancement strategy generation

3.3.1. Motivation

As previously described, most existing machine learning-based vulnerability detection approaches are predominantly data-driven. They rely on the assumption that neural network models can automatically learn task-relevant semantics from intermediate representations (*e.g.*, graph structures) (Lakadawala et al., 2024; Luo et al., 2024; Zhuang et al., 2021). However, in practice, these methods often overlook domain-specific semantics of smart contracts and lack explicit guidance for the detection models, which results in limited semantic understanding. Furthermore, Liu et al. (2021a,b) attempt to incorporate expert knowledge by directly concatenating expert-defined features with learned graph representations. Such feature-level fusion methods, which do not operate directly on intermediate representations, often suffer from insufficient structural integration and may fail to fully utilize the complementary advantages of heterogeneous information.

To address the problem, we assign different importance coefficients to the nodes of the SRCG based on domain knowledge of vulnerability-related semantics for reflecting its semantic salience. For example, control-sensitive operations are given higher importance. This reweighting mechanism highlights semantically critical nodes and guides the model to more meaningful representations during downstream learning.

3.3.2. Importance coefficient assignment

We present Algorithm 1 to demonstrate how the embedding enhancement strategy is enabled on the SRCG through node-level importance labeling based on different types of vulnerabilities. This algorithm helps the detection model distinguish and emphasize semantically critical local structures during representation learning.

A typical feature of reentrancy vulnerabilities is the presence of a loop involving external calls at the graph level. Specifically, a *call.value()*

Algorithm 1 Assigning node importance in the SRCG.

```
Require: SRCG g, a target vulnerability type t, the high importance h,
    the moderate importance m, and the normal importance o
Ensure: a map mp from nodes to their importance
 1: if t is Reentrancy or Infinite Loop then
 2:     retrieve vulnerability-related subgraphs subs on g
 3:     for each subgraphs sub in subs do
 4:         retrieve vulnerability-related risky nodes rn in sub
 5:         $mp \leftarrow mp \cup \{\langle n, h\rangle \mid n \in rn\}$
 6:         res := remove rn from all nodes of sub
 7:         $mp \leftarrow mp \cup \{\langle n, m\rangle \mid n \in res\}$
 8:     end for
 9:     $mp \leftarrow mp \cup \{\langle n, o\rangle \mid n \in$ nodes in g and $n \notin$ nodes in subs$\}$
10: end if
11: if t is Timestamp Dependence then
12:     retrieve data-flow paths dfps on g with block.timestamp as the
        source
13:     $mp \leftarrow mp \cup \{\langle n, h\rangle \mid n \in$ nodes in dfps$\}$
14:     conds := get control nodes that do not depend on block.timestamp
        but have control over the executions of the nodes in dfps
15:     $mp \leftarrow mp \cup \{\langle n, m\rangle \mid n \in conds\}$
16:     $mp \leftarrow mp \cup \{\langle n, o\rangle \mid n \in$ nodes in g and $n \notin$ nodes in dfps$\}$
17: end if
```

leads to a fallback function, which in turn re-enters the contract's entry point, potentially allowing repeated invocations. Therefore, for the reentrancy vulnerabilities, we firstly retrieve vulnerability-related subgraphs (*i.e.*, the specific loops) on the SRGC (Line 2). Then for each of the subgraph, we find risky nodes, including action nodes, fallback nodes, and basic nodes related to the contract's balance (Lines 3–4). Considering that the correlations between the risky nodes and reentrancy vulnerabilities, we assign the nodes the high importance (line 5). Although not directly associated with the vulnerabilities, the other nodes in the loops contribute contextual information. Hence, we assign the nodes the moderate importance (Line 7). Finally, the remaining nodes in the SRCG are assigned the normal importance (Line 9).

Infinite loop vulnerabilities are often characterized by the existence of self-sustaining control-flow cycles at the graph level, where execution can continue indefinitely without reaching a termination point. Although the loop structures resemble those found in reentrancy vulnerabilities, they stem from different semantic conditions and therefore require a distinct detection strategy. Specifically, at Line 2, we focus

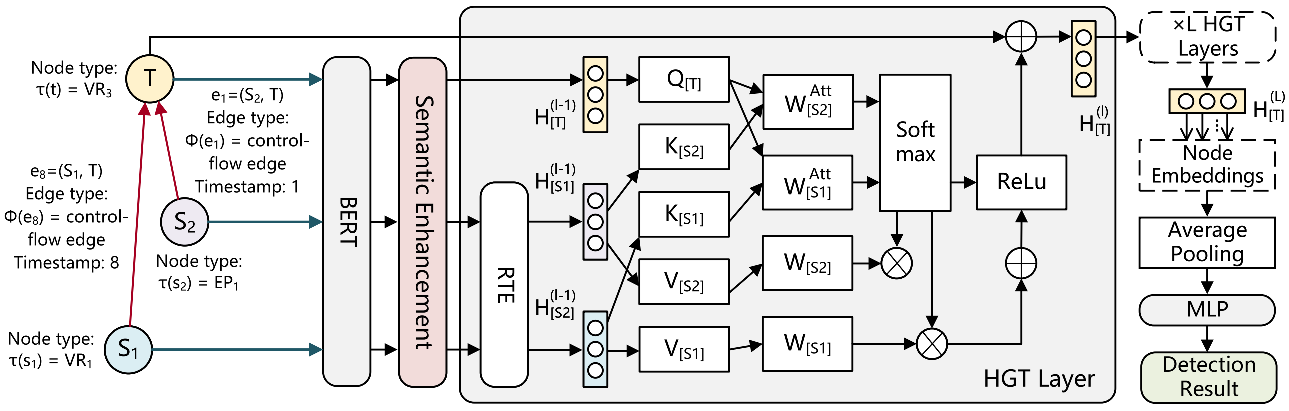


Fig. 3. Deep neural network architecture for vulnerability detection in SmartScope.

on identifying subgraphs that represent loop constructs or cyclic inter-procedural calls among multiple functions. Then, at Line 4, the risky nodes of infinite loop vulnerabilities include control nodes, action nodes with the invocations of customized functions, and basic nodes with data dependency on *if()* statements.

In contrast to the two types of vulnerabilities above, which require analyzing loop-related structures, timestamp dependence vulnerabilities involve the misuse of external time sources, requiring data-flow analysis to trace their influence on critical operations. Specifically, we first retrieve the data-flow paths originating from *block.timestamp* on the SRCG (Line 12). If conditional statements does not have a direct data dependency on the timestamp variables but have control over the executions of the nodes within the retrieved data-flow paths, we maintain that it is still related to vulnerability analysis. Therefore, we assign the moderate importance on the corresponding control nodes (Lines 14–15). Similarly, the remaining nodes in the SRCG are assigned the normal importance (Line 16).

The map of node importance assigned by Algorithm 1 is fed into the downstream graph representation module to guide the neural network in focusing on semantically important local regions.

3.4. Vulnerability detection

3.4.1. Motivation

Based on the previous steps, we construct the SRCG and the corresponding strategy for local semantic enhancement. We next encode the core semantics embedded in the graph for enabling the detection of potential vulnerabilities. To highlight the varying contributions of different types of nodes and edges to the detection, we choose to use the heterogeneous graph representation technique. Faced with various widely-used heterogeneous graph representation learning models, we adopt the heterogeneous graph transformer (*i.e.*, HGT Wang et al., 2019) model for the following three reasons:

(1) HGT does not rely on manually designed meta-paths. Due to the semantic complexity of Solidity code and the variety of inter-element relations, manually defining meta-paths, as done in heterogeneous graph attention network (HAN) (Wang et al., 2019), is often fails to capture the full range of meaningful patterns. In contrast, HGT can automatically capture heterogeneous interactions through type-specific transformations and attention mechanisms, making it more suitable for capturing the rich structural and semantic information in the SRCG.

(2) HGT is capable of encoding temporal information, *i.e.*, timestamps in our SRCG, which is critical for capturing execution semantics in program analysis. In smart contracts, this temporal aspect is even more pronounced, as certain vulnerabilities directly arise from the misuse of time-related features, *e.g.*, timestamp dependence. Accurate encoding of such temporal information is crucial for detecting vulnerabilities. HAN falls short in capturing the dynamic aspects of program behavior.

(3) HGT has demonstrated strong architectural capabilities and has been extensively validated across diverse domains, including vulnerability detection (Sun et al., 2025b), tiny object tracking (Xu et al., 2024), and bioinformatics (Ma et al., 2023). Compared to many recent heterogeneous GNN variants, HGT achieves a better balance among stability, expressiveness, and ease of integration into complex pipelines. In the context of safety-critical tasks such as smart contract analysis, HGT's robustness and reliability make it a more suitable choice than models that prioritize novelty alone.

As shown in Fig. 3, we design a deep neural network architecture for detecting vulnerabilities in smart contracts. The architecture is divided into three components: node feature vectorization, heterogeneous graph embedding, and MLP-based classification.

3.4.2. Node feature vectorization

As depicted in the left of Fig. 3, this component vectorizes the feature of each node on the SRCG to obtain its embedding. As depicted in Section 3.2.4, the node feature is present as a tuple of its identifier, its type, and the element of the code within it. We firstly encode each node by feeding the concatenation of its type and the corresponding lexical element into the pre-trained BERT model (Devlin et al., 2019), and then use the resulting output as its initial semantic representation. For example, the initial representation of node T is calculated by

$$h^{(0)} = \text{BERT}([t; e]), \tag{1}$$

where t is the type of the node, e is the element of the node, and $h^{(0)}$ is the output of the BERT model. We next enhance the representations of all nodes by the layer of Semantic Enhancement in Fig. 3 based on the output of Algorithm 1. Given that the importance assigned to the node T is m, the output of the layer is calculated by

$$H^{(0)} = m \circ h^{(0)}, \tag{2}$$

where $\circ$ is the element-wise multiplication, and $H^{(0)}$ is one of the initial inputs of the HGT model. The specific value of the importance score (*e.g.*, m) will be determined empirically through experiments. Finally, the generated vectors for all nodes serve as the inputs for the subsequent component.

3.4.3. Heterogeneous graph embedding

We leverage the HGT to encode the SRCG and generate final embeddings for each node. In the following, we provide a brief overview of the overall procedure of the HGT.

As shown in the middle part of Fig. 3, the relative temporal encoding (*i.e.*, the RTE layer) technique in the HGT is firstly adopted to obtain the temporal augmented representations. To effectively capture the heterogeneity of the left subgraph, the HGT learns the attention value of every pair of meta relations between a target node T and its neighbor source nodes S_1 and S_2, based on the representations. The output of the (l)-th

HGT layer is denoted as $H^{(l)}$, which is also the input of the $(l+1)$-th layer. By stacking L layers, we obtain the node representations of the entire graph $H^{(L)}$, which can be incorporated into downstream tasks.

Heterogeneous Mutual Attention. At the $l-1$ layer of the HGT, the input vectors of the nodes T, S_1, and S_2 correspond to H^{l-1}, $\mathrm{H}^{l-1}_{[S_1]}$, and $\mathrm{H}^{l-1}_{[S_2]}$ respectively. After that, the HGT computes attention values between the target node and its source neighbors. The query and key projections of nodes T, S_1, and S_2 are obtained via linear transformations (*i.e.*, Q and K). The attention matrices $W^{\mathrm{ATT}}_{[S1]}$ and $W^{\mathrm{ATT}}_{[S2]}$ preserve relation-specific characteristics during attention computation. The heterogeneous attention between different node types is computed as follows. $H^{(l-1)}_{[S1]}$ and $H^{(l-1)}$ are projected through linear layers to the i^{th} head key vector K^i and query vector Q^i respectively, and the parameters are not shared across heads to maximize the distribution difference. The attention of nodes T and S_1 is calculated by:

$$\mathrm{Att}(T,S_1) = \mathrm{Softmax}\big(\mathrm{Concat}_{i\in[1,h]}\big(\mathrm{Att}^i(T,S_1)\big)\big), \tag{3}$$

$$\mathrm{Att}^i(T,S_1) = \mathrm{Linear}(K^i(H^{(l-1)}_{[S1]}))\, W^{\mathrm{ATT}}_{[S1]}\, \mathrm{Linear}(Q^i(H^{(l-1)})), \tag{4}$$

where h is the number of attention heads, the matrix $W^{\mathrm{ATT}}_{[S1]}$ is distinct to capture the semantic relations between the specific node type pairs. The attention between T and S_1 is then generated by applying softmax to all neighboring nodes to ensure that the weights sum up to 1.

Heterogeneous message passing. The message from node S_1 to T incorporates the edge dependency using a distinct matrix $W_{[S1]}$:

$$\mathrm{Message}(S_1) = \mathrm{Softmax}(\mathrm{Concat}_{i\in[1,h]}\mathrm{Message}^i(S_1)), \tag{5}$$

$$\mathrm{Message}^i(S_1) = \mathrm{Linear}(K^i(H^{(l-1)}_{[S1]}))W_{[S1]}, \tag{6}$$

where the i-th message head projects the source node representation $H^{(l-1)}_{[S1]}$ to the i-th message vector K^i. K^i is then multiplied with matrix $W_{[S1]}$ to incorporate edge dependency. The final message passing step is to concatenate all heads messages to get $\mathrm{Message}(S_1)$.

Target-specific aggregation. The output of HGT is finally generated by using the attention values to aggregate messages from all types of source nodes. The updated $H^{(l)}$ is calculated as follows:

$$H^{(l)} = \sum_{\forall S\in N(T)} (\mathrm{Att}(T,S)\cdot \mathrm{Message}(S)), \tag{7}$$

where $\sum$ aggregates the message to the target node T from all its neighbors $N(T)$ (*i.e.*, source nodes) with the corresponding attention values.

The target node T goes through L HGT layers to create the embedding vector $H^{(L)}$. Such a mechanism enhances the representation learning of the SRCG of smart contracts by capturing complex heterogeneous interactions through transformer architecture.

3.4.4. MLP-based classification

In the final stage of the vulnerability detection task, which essentially reduces to graph-level classification, an efficient yet effective classifier is required to ensure the accuracy of SmartScope. To produce a unified representation of the SRCG, we apply the average pooling over all node embeddings. This parameter-free operation efficiently aggregates multi-type node information into a compact feature vector. Given that the pooled vector is compact and semantically rich, a lightweight MLP suffices for the final classification without introducing unnecessary complexity. Despite its simplicity, the MLP provides strong nonlinear modeling capacity and is widely adopted in downstream graph tasks, so it is widely used in existing work of smart contract detection (Liu et al., 2021a; Luo et al., 2024). The final classification is performed by feeding the aggregated representation into this MLP. The detailed configuration of our MLP-based classifier is listed Table 3. The model is trained using the cross-entropy loss, a standard choice for binary classification tasks.

Table 3
Parameters of our MLP-based classifier.

Designed Layer	Dimension	Activation	Using Dropout
Input Layer	128	none	no
Hidden Layer1	256	ReLU	yes
Hidden Layer2	64	ReLU	yes
Output Layer	1	Sigmoid	no

4. Experimental evaluation

To evaluate the effectiveness of SmartScope, we seek to answer the following five questions:

- RQ1: How should node importance be determined to achieve effective smart contract vulnerability detection?
- RQ2: Can SmartScope detect the three types of smart contract vulnerabilities compared to existing deep learning-based methods?
- RQ3: How does SmartScope perform compared to state-of-the-art conventional smart contract vulnerability detection methods?
- RQ4: What is the contributions of the essential modules to the overall performance of SmartScope?
- RQ5: What is the time overhead of SmartScope for the detection?

4.1. Experimental setup

4.1.1. Implementation

We implement a prototype of SmartScope. Specifically, to collect the required information from Solidity code of smart contracts, we perform static analysis using the built-in interfaces provided by the Slither framework (Feist et al., 2019). We then construct the SRCG based on an intermediate representation called SlithIR. The graph learning and MLP-based classification processes are based on PyTorch. The experiments are conducted on a machine with AMD Ryzen 5 7600X 6-Core Processor 4.70 GHz CPU, 128GB memory, Ubuntu 22.04 LTS (64bit) and NVIDIA GeForce RTX 3080 Ti (12GB) GPU.

4.1.2. Dataset

To build a suitable dataset, we have made substantial efforts. On the one hand, we observe that the labels in the real-world dataset (Qian et al., 2023b) are sometimes inconsistent with the definitions of vulnerabilities. For example, as the smart contract shown in Listing 1, since miners can manipulate *block.timestamp*, they may influence the condition at Line 6, which in turn directly affects the execution of the *send* function at Line 7. However, this contract is labeled as vulnerability-free in the dataset. Therefore, using such data may lead to inaccurate training or evaluation of detection tools. On the other hand, as explained in previous work, the intersection of multiple detection tools lacks reliability due to the inherent inaccuracies of each tool.

We choose to build the dataset used in our experiments by combining manual analysis and LLM generation. On the one hand, for the three widely-used datasets (Liu et al., 2021b; Luo et al., 2024; Qian et al., 2023a), with over 40K smart contracts respectively, we invite 2 experts with at least 2 years of blockchain experience to manually label 100 samples for each type of vulnerability we detect.

On the other hand, we adopt a hybrid approach that combines advanced LLMs with manual checking to efficiently expand the size of our dataset. Considering that only a small portion of the unlabeled contracts in the above datasets actually contain vulnerabilities, manually analyzing each contract would be extremely time-consuming. Therefore, we adopt a prompt-based generation strategy using an LLM. Specifically, we first select a state-of-the-art and widely-used LLM, GPT-4, for our task. We then provide the LLM with historical smart contract code sourced from real-world examples as context, and construct targeted prompts with explicit requirements, such as vulnerability type, Solidity version, function structure, and control-flow features, to guide the generation of semantically valid and contextually relevant contracts. To ensure that the overall structure and complexity of the generated contracts are comparable to real-world smart contracts:

```
contract BirthdayGift {
    address public recipient;
    uint public birthday;
    function Take() {
        if (msg.sender != recipient) throw;
        if (block.timestamp < birthday) throw;
        if (!recipient.send(this.balance)) throw;
        return;
    }
}
```

Listing 1. Example of a smart contract mislabeled in the open-source dataset.

Table 4
Composition of the dataset in our experiments.

Category	Reentrancy	Time Dependence	Infinite Loop	Sum
Vulnerable	601	1085	611	2297
Clean	728	1346	702	2776

- We perform double-checking of each contract's semantic correctness by the two experts mentioned above, ensuring the intended vulnerability is logically valid and realistically embedded.
- We use the off-the-shelf tool, Slither, to verify the syntactic correctness and static analyzability of all generated smart contracts, all of which can be successfully compiled.

This approach has been explored in recent literature and allows for the generation of diverse, vulnerability-grounded smart contracts (Chen et al., 2025). The LLM-generated contracts serve as a valuable supplement to real-world data.

Our final dataset is listed in Table 4. To the best of our knowledge, its size is larger than those used in many existing representative studies (Luo et al., 2024; Nguyen et al., 2023). Note that we perform per-vulnerability binary classification for each type, and construct datasets with relatively balanced numbers of vulnerable and clean contracts.

4.1.3. Parameter setting

We employ the Adam optimizer to train our network and perform a grid search to identify the optimal hyperparameter settings: the learning rate l is tuned amongst {0.0001,0.00005,0.00001,0.0002, 0.0005,0.001}, the dropout rate d is searched in {0.1, 0.2, 0.3, 0.4, 0.5}, and and batch size b is in {8, 16, 32, 64, 128}. As the result, the default parameters of model are as follows: $l = 0.00005$, $b = 32$, and $d = 0.5$. For each type of vulnerability, we use 5-fold cross-validation to ensure that the train set represents all patterns involved in the vulnerability and report the average results. Each fold maintains an approximately equal amount of vulnerable and clean data.

4.1.4. Evaluation metric

We define the evaluation metrics for smart contract vulnerability detection as follows: A true positive (TP) occurs when a contract with a specific vulnerability is correctly predicted as having that vulnerability. A true negative (TN) refers to a secure contract that is correctly predicted as non-vulnerable. A false positive (FP) occurs when a contract that does not have a specific vulnerability is incorrectly predicted as having it. A false negative (FN) refers to a vulnerable contract that has a specific vulnerability but is either missed entirely or misclassified as another type. Based on these definitions, we evaluate the detection model using the four metrics: Precision, Recall, Accuracy, and F_1-score.

4.2. RQ1: importance coefficient assignment

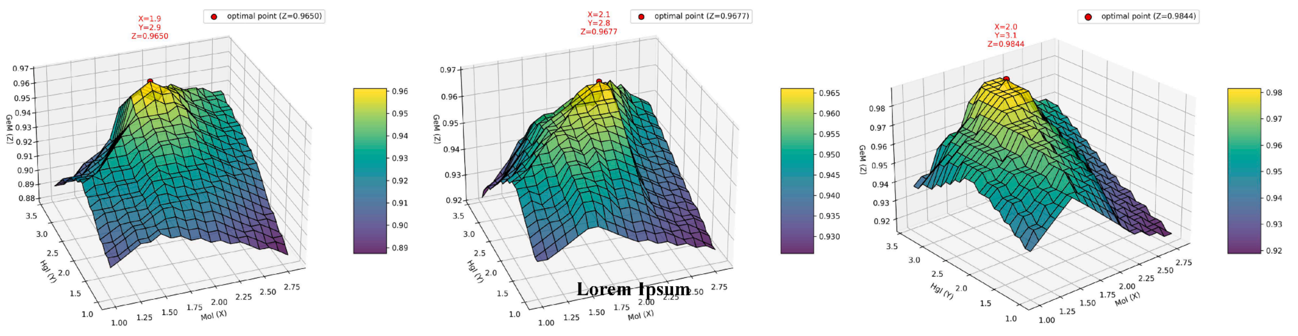
Before performing comprehensive experiments, we first determine the most appropriate importance coefficients for different types of vulnerabilities. As explained in Section 3.3.2, we define three levels of importance coefficients, namely high, moderate, and normal. For the normal level, we do not apply any specific weighting method, *i.e.*, assigning a default importance coefficient of 1. For the moderate and high levels, given the large number of possible combinations, we perform a two-stage experiments to progressively determine their optimal values. In the first stage, we identify a reasonable range of values for the importance coefficients. In the second stage, we conduct a finer-grained evaluation within this range to determine the specific values that yield the best detection performance. Furthermore, for each group of coefficients, we calculate the F_1-score and Accuracy, and use their geometric mean $GeM = \sqrt{F_1\text{-score} \times \text{Accuracy}}$ as a unified metric to reflect overall detection performance.

As listed in Table 5, we empirically select eight representative combinations of the moderate and high levels to evaluate the performance of SmartScope. It can be observed that the *GeM* reaches its peak when the combination is 2:3, and the performance degrades more noticeably as the ratio deviates further from this point. In other words, the optimal ratio between the moderate and high levels is likely to be around 2:3. Theoretically, these two values are designed to enhance the relative influence of certain components in the SRCG without introducing instability or imbalance into the detection model. However, excessively large values (*e.g.*, 6:8, 7:10, and 9:13) may suppress the contribution of other contextual features. Therefore, in the second-stage experiment, we restrict the value ranges of *Mol* and *Hgl* to [1, 3] and [1, 3.5] respectively, to further explore the optimal configuration.

Based on the results from the first-stage experiment, we perform a grid search to identify the optimal importance coefficient respectively: the moderate level *Mol* is searched in {1.0, 1.1, 1.2, ..., 3.0}, the high level *Hgl* is tuned in {1.0, 1.1, 1.2, ..., 3.5}. The experimental results are drawn in Fig. 4, where the optimal point for each type of vulnerability is highlighted in red. Specifically, for the detection of the reentrancy vulnerability, the maximum value of *GeM* is 0.9650, achieved when *Mol* is 1.9 and *Hgl* is 2.9. When *Mol* and *Hgl* are set to other values, SmartScope's performance drops gradually, validating the

Table 5
Detection performance of SmartScope with different combinations of the moderate and high levels for each type of vulnerability.

Vulnerability	1:1	1:2	2:3	3:3.5	4:5	6:8	7:10	9:13
Reentrancy	0.8985	0.9220	**0.9571**	0.9380	0.9188	0.8879	0.8570	0.8362
Time dependence	0.9332	0.9485	**0.9589**	0.9441	0.9211	0.8983	0.8772	0.8602
Infinite loop	0.9381	0.9534	**0.9805**	0.9727	0.9416	0.9226	0.8953	0.8762



(a) For reentrancy (b) For timestamp dependence (c) For infinite loop

Fig. 4. Detection performance of SMARTSCOPE with different importance coefficient assignments for each vulnerability.

effectiveness of the selected configuration. Similarly, for the detection of the timestamp dependence vulnerability, the maximum value of *GeM* is 0.9677, obtained when *Mol* is 2.1 and *Hgl* is 2.8. For the detection of the infinite loop vulnerability, the maximum value of *GeM* is 0.9844, reached when *Mol* is 2.0 and *Hgl* is 3.1. Consequently, we use the optimal importance coefficients identified above to detect the corresponding vulnerabilities.

4.3. RQ2: comparison with existing deep learning-based methods

To evaluate SMARTSCOPE's effectiveness, we compare its performance to other deep learning-based vulnerability detection methods, which include:

- TMP (Zhuang et al., 2021): A temporal message passing network for smart contract vulnerability detection, which performs graph-level classification by passing messages along temporally ordered edges and aggregating node representations through a readout function.
- AME (Liu et al., 2021a): An attentive multi-encoder network designed for smart contract vulnerability detection, which combines local expert patterns encoded by MLPs with global graph features extracted via a TMP network to generate the final prediction.
- HAN (Luo et al., 2024): A heterogeneous attention network that can be applied to smart contract vulnerability detection by capturing semantic information across multiple node and edge types through hierarchical attention mechanisms on heterogeneous contract graphs.
- GCN (Zhuang et al., 2021): A neural network architecture for processing graph-structured data, which captures structural dependencies in code representations, *e.g.*, CFGs, to assist in detecting smart contract vulnerabilities.
- HetGNN (Ye et al., 2023): A heterogeneous graph neural network used in domains such as supply chain vulnerability detection, which learns from diverse node and edge types and shows potential for modeling complex smart contract structures.

These methods are representative of typical vulnerability detection applications based on their respective models, and are widely adopted as baselines in recent studies on smart contract vulnerability detection. In our experiment with HAN, due to the lack of detailed meta-path descriptions in SCVHunter, we manually construct meta-paths by exhaustively pairing node types in the SRCG to capture underlying semantic relationships. These meta-paths are then used as inputs to the HAN model.

The experimental results are listed in the upper part of Table 6. Overall, SMARTSCOPE shows superior performance compared to other deep learning-based methods in detecting the three types of vulnerabilities in smart contracts. Specifically, for reentrancy, timestamp dependence, and infinite loop vulnerabilities, SMARTSCOPE improves the detection accuracy by 4.62 %, 3.11 %, and 4.18 %, respectively, compared to HAN, the best-performing model in the experiment. This suggests not only that applying the embedding enhancement strategy into graph embeddings can effectively guide the detection model, but also that HGT is more capable than HAN of capturing the semantics of smart contracts in this context. After that, as a representative approach that integrates expert knowledge with graph features, AME still underperforms compared to our tool. This demonstrates that directly modifying the node representations based on the domain knowledge of vulnerability-related semantics is more effective than combining the graph embedding with local expert patterns about vulnerabilities by feature vector concatenation. Furthermore, we also notice that heterogeneous graph neural networks (*e.g.*, HetGNN) outperform their homogeneous counterparts (*e.g.*, GCN) in terms of overall effectiveness. This suggests that heterogeneous GNNs can better capture and represent the behavioral semantics of smart contracts, thereby enabling more accurate detection of vulnerabilities.

Table 6
Effectiveness comparison of SMARTSCOPE and existing methods, where "-" means the method does not support detecting this type of vulnerability.

Method	Reentrancy				Timestamp Dependence				Infinite Loop			
	Pre(%)	Rec(%)	Acc(%)	F1(%)	Pre(%)	Rec(%)	Acc(%)	F1(%)	Pre(%)	Rec(%)	Acc(%)	F1(%)
TMP	83.61	81.60	83.77	0.82	84.89	85.27	86.19	0.85	85.60	84.92	85.93	0.85
AME	91.58	77.78	85.93	0.84	87.78	86.61	88.25	0.87	86.92	89.68	88.61	0.88
HAN	93.39	89.68	91.98	0.91	94.09	92.41	93.80	0.94	93.02	95.23	94.30	0.94
GCN	81.06	79.30	80.73	0.80	78.81	83.04	81.86	0.81	82.03	83.33	83.27	0.82
HetGNN	93.91	86.40	90.94	0.90	90.75	91.56	91.75	0.91	91.74	88.10	90.49	0.90
SmartCheck	57.02	51.59	58.17	0.54	60.87	62.22	63.92	0.62	54.35	39.68	55.13	0.46
Oyente	66.67	53.54	64.91	0.59	54.02	62.67	57.94	0.58	-	-	-	-
Securify	68.63	65.42	73.96	0.67	-	-	-	-	-	-	-	-
Mythril	71.57	69.52	76.98	0.71	55.68	89.33	62.06	0.69	74.74	56.35	69.96	0.64
Slither	84.16	70.83	80.75	0.77	73.29	52.44	69.07	0.61	68.57	57.14	66.92	0.62
SMARTSCOPE	**96.03**	**96.80**	**96.60**	**0.96**	**97.30**	**96.00**	**96.91**	**0.97**	**99.19**	**97.62**	**98.48**	**0.98**

Table 7
Ablation results on the three types of vulnerability detection tasks.

Method	Reentrancy				Timestamp Dependence				Infinite Loop			
	Pre(%)	Rec(%)	Acc(%)	F1(%)	Pre(%)	Rec(%)	Acc(%)	F1(%)	Pre(%)	Rec(%)	Acc(%)	F1(%)
w/o SRCG	88.50	79.37	85.17	0.84	88.08	75.56	83.92	0.81	95.65	87.30	91.98	0.92
w/o BIAS	94.64	84.13	90.11	0.89	93.30	92.89	93.61	0.93	92.97	94.44	94.07	0.94
w/o HGT	64.36	51.59	63.12	0.57	55.12	62.22	58.97	0.58	65.93	47.62	63.12	0.55
w/o AN	87.88	92.71	90.57	0.90	91.96	91.56	92.37	0.92	95.31	96.83	96.20	0.96
w/o CN	87.69	91.18	89.81	0.89	89.96	91.56	91.34	0.91	90.55	91.27	91.25	0.91
w/o FN&FBE	82.09	88.00	85.28	0.85	95.93	94.22	95.46	0.95	97.56	95.24	96.58	0.96
w/o DFE	89.92	92.78	91.70	0.91	92.89	92.89	93.40	0.93	92.80	92.06	92.78	0.92
SmartScope	**96.03**	**96.80**	**96.60**	**0.96**	**97.30**	**96.00**	**96.91**	**0.97**	**99.19**	**97.62**	**98.48**	**0.98**

4.4. RQ3: comparison with state-of-the-art conventional detection tools

We benchmark our tool against existing non-deep-learning vulnerability detection approaches, which include:

- SmartCheck (Tikhomirov et al., 2018): An extensible static analysis tool for discovering vulnerabilities in smart contracts.
- Oyente (Luu et al., 2016): A symbolic execution tool for vulnerability detection in smart contracts by exploring their control-flow graphs.
- Securify (Tsankov et al., 2018): A formal-verification-based tool that can be used to identify vulnerabilities in smart contracts with respect to given properties.
- Mythril (Mueller, 2017): A symbolic-execution-based security analysis tool for detecting vulnerabilities in smart contracts.
- Slither (Feist et al., 2019): A static analysis framework for detecting vulnerable Solidity code based on an intermediate representation of SlithIR.

These tools are collected from top-tier conferences, top journals, and related publications in security or software engineering research, and commonly used as baselines. Even Mythril, which was proposed in 2017, has been continuously updated and maintained since then. The experimental results are listed in the lower part of Table 6.

For the results of reentrancy vulnerability, we notice that SmartScope achieves better detection performance than traditional tools on our dataset. Specifically, as the best-performing conventional detection tool, Slither achieves a detection Accuracy of 80.75 %. In comparison, our tool reaches an Accuracy of 96.60 %. Conventional tools perform poorly primarily because they rely heavily on simple and static patterns for detecting vulnerabilities. For example, Mythril detects reentrancy by checking whether a *call.value()* invocation is followed by any internal function call. Furthermore, these methods fail to capture the fine-grained data and control dependencies in smart contracts (Liu et al., 2021b). By contrast, SmartScope achieves superior results by leveraging heterogeneous graph embeddings with local semantic enhancement to automatically extract critical features from fine-grained smart contract models.

For the results of timestamp dependence vulnerability, since Securify does not support the detection of this vulnerability, we directly compare SmartScope with SmartCheck, Oyente, Mythril, and Slither. Specifically, we can see that SmartScope still outperforms the conventional methods on all four metrics. SmartScope achieves an Accuracy of 96.91 % and an F_1-score of 0.97, significantly surpassing Smartcheck (63.92 % and 0.62), Oyente (57.94 % and 0.58), Mythril (62.06 % and 0.69), and Slither (69.07 % and 0.61) respectively. Their poor performance may stem from the fact that most conventional tools address timestamp dependence vulnerabilities by merely checking for the presence of *block.timestamp* in a function, without assessing whether it actually influences any critical operations (Liu et al., 2021a). By modeling and embedding the contextual semantics of *block.timestamp*, SmartScope accurately identifies the timestamp dependence vulnerabilities.

For the results of infinite loop vulnerability, we only compare our tools with SmartCheck, Mythril, and Slither, as they are the only ones that support the detection of the vulnerability. We observe that SmartScope also outperforms the three tools on all metrics. Moreover, we also notice that compared to the other two types of vulnerabilities, SmartScope achieves the best detection performance for infinite loop vulnerabilities. In comparison, traditional tools perform the worst in detecting infinite loop vulnerabilities among the three detection tasks. This suggests that it is difficult to detect such vulnerabilities using fixed patterns alone, whereas our heterogeneous graph learning method can uncover hidden vulnerability semantics.

4.5. RQ4: ablation study

4.5.1. Overall results

As depicted before, there are three essential modules in the design of SmartScope, including SRCG construction, importance coefficient assignment, and HGT network. To demonstrate the effectiveness of the modules in the detection, we conduct three ablation studies and the experimental results are listed in Table 7. The table contains three alternatives:

- Following the graph construction strategy of SCVHunter (Luo et al., 2024), we simplify the SRCG by removing control nodes, *e.g.*, *if()* (*i.e.*, *w/o* SRCG).
- We do not add the importance coefficient assignment to initial node representations (*i.e.*, *w/o* BIAS).
- We do not use the HGT network but directly fuse the final embeddings of nodes by the average pooling (*i.e.*, *w/o* HGT).

As shown at the third row of Table 7, SmartScope achieves better detection performance when using the original SRCG compared to the SCVHunter-used model. For example, in the representative task of timestamp dependence vulnerability detection, the standard SmartScope significantly outperforms the first alternative, achieving improvements of 9.22 %, 20.44 %, 12.99 %, and 0.16 in terms of Precision, Recall, Accuracy, and F_1-score, respectively. These results indicate that, compared to the approach that does not explicitly model control nodes, the fine-grained control semantics in the SRCG are important to detect smart contract vulnerabilities, especially for timestamp dependence.

As shown at the fourth row of Table 7, SmartScope still outperforms the tool without assigning importance coefficients on the whole. Among the three types of vulnerability detection tasks, assigning coefficients brings the greatest improvement in the detection of reentrancy vulnerability. Specifically, the standard SmartScope and the second alternative approach achieve comparable Precision, but the former exhibits significantly higher Recall than the latter. This phenomenon indicates that introducing the bias in our work helps enhance the semantic saliency of subgraphs related to various vulnerabilities in SRCG, particularly for the reentrancy in Table 7, thereby guiding the detection model to find vulnerabilities more effectively.

As shown at the fifth row of Table 7, SmartScope significantly outperforms the third alternative in all metrics. Without the comprehensive

```
1  function claimJackpot() external {
2      ...
3      uint ts = block.timestamp;
4      uint reward = 0;
5      if ((ts % 100 == 42) && (ts / 60 % 2 == 1) && ((ts >> 2)
           % 5 == 3)) {
6          reward = jackpotPool / 100;
7          require(reward > 0, "No reward available");
8          jackpotPool -= reward;
9          payable(msg.sender).transfer(reward);
10     }
11     jackpotPool -= amount;
12     balances[msg.sender] = 0;
13     ...
14 }
```

Listing 2. Example of the time dependence vulnerability in a real-world smart contract.

graph representations provided by the HGT, the downstream MLP-based classifier fails to effectively detect vulnerabilities in smart contracts. In other words, HGT is capable of effectively capturing the heterogeneous semantic information in the SRCG, thereby facilitating the detection of various vulnerabilities in smart contracts.

4.5.2. *Contribution of individual node and edge types*

We conduct another ablation experiments by selectively masking specific node and edge types in the SRCG to assess their individual contributions to vulnerability detection. To ensure the structural integrity and semantic continuity of the graph, we adopt a masking strategy rather than removing nodes or edges outright. Specifically, we retain basic nodes and control-flow edges as the core backbone of the SRCG. When masking a particular node type, we relabel those nodes as basic nodes instead of deleting them, thereby preserving graph connectivity while neutralizing their semantic role. Similarly, when masking an edge type, we relabel those edges as control-flow edges to prevent graph fragmentation. This allows us to isolate the contribution of each node or edge type while keeping the overall topology and information flow of the graph intact. We therefore design four alternatives:

- We label action nodes as basic nodes (*i.e.*, *w/o* AN).
- We label control nodes as basic nodes (*i.e.*, *w/o* CN).
- We simultaneously label fallback nodes as basic nodes, and fallback edges as control-flow edges, because both of them are related to the fallback mechanism (*i.e.*, *w/o* FN&FBE).
- We label data-flow edges as control-flow edges (*i.e.*, *w/o* DFE).

As shown in rows 6-9 of Table 7, removing any single node type or edge type from the SRCG leads to a noticeable drop in the detection performance of our tool. Specifically, (1) for the reentrancy vulnerability detection, removing the FN&FBE leads to a notable F_1-score drop from 0.96 to 0.85, indicating that the fallback mechanism has a critical role in detecting such vulnerability. In contrast, the DFE shows minimal impact. (2) Timestamp dependence vulnerability detection is particularly sensitive to the removal of AN and CN, while FN&FBE contributes relatively less. It is known that the vulnerabilities are closely related to conditional statements and operations performed after the condition is triggered, so action nodes and control nodes have greater contributions. (3) For infinite loop vulnerability detection, removing CN and DFE leads to relatively notable F_1-score drops from 0.98 to 0.91 and 0.92, respectively, indicating that both control-flow and data-flow edges play important roles in capturing loop-related behaviors. In contrast, the absence of AN and FN&FBE has little effect on performance, suggesting that action semantics and fallback mechanisms are less critical for detecting this type of vulnerability.

4.5.3. *Case study*

We present Listing 2 to further demonstrate the effectiveness of our graph construction strategy. This function contains a time dependence vulnerability, as the reward distribution logic relies on specific conditions derived from *block.timestamp* at Line 3. Since miners can manipulate the timestamp within a limited range, an attacker may repeatedly call the function at carefully chosen times to satisfy the condition and extract the reward unfairly.

Based on the above analysis, it is evident that accurately modeling the conditional statement on Line 5, along with its surrounding context, is crucial for detecting this vulnerability. However, the vulnerability is not detected when applying the graph modeling approach of the advanced tool named SCVHunter (*i.e.*, *w/o* SRCG in Table 7). In contrast, the vulnerability is correctly identified when using the SRGC. The fine-grained modeling strategy adopted by SRCG effectively captures rich semantics about the variable named *block.timestamp* and the predicate of the conditional statement, ensuring the vulnerability's semantics are preserved and accurately detected.

4.6. *RQ5: time overhead*

4.6.1. *Overall comparison*

We conduct an experiment to compare the time overhead of SmartScope with the 10 baseline tools. To ensure a fair and accurate evaluation, all background processes are disabled, and the experiments are performed in a clean environment. Each tool is run on the entire dataset, and we compute the average time required to analyze a single contract.

Fig. 5 presents the average time cost required by each tool. The results demonstrate significant variation in efficiency across the 10 baseline tools and SmartScope. Overall, SmartScope achieves one of the lowest time costs (0.34 seconds), indicating its high efficiency in vulnerability detection. Compared to the conventional analysis tools, such as Mythril (39.27 seconds), Oyente (18.39 seconds), and Securify (11.06 seconds), SmartScope is orders of magnitude faster, which highlights its practical advantage in large-scale analysis scenarios. The deep learning-based methods, like TMP (0.29 seconds), GCN (0.29 seconds), and AME (0.32 seconds), also exhibit low overhead, suggesting that these approaches generally offer better scalability than conventional static analysis methods. However, SmartScope remains competitive with these lightweight baselines, while providing more comprehensive analysis capabilities, as demonstrated in Table 6.

In summary, the results above indicate that SmartScope effectively balances accuracy and scalability, making it well-suited for practical deployment in large-scale smart contract auditing.

4.6.2. *Component analysis*

We evaluate the time overhead of SmartScope in detecting a smart contract detailedly. As the experimental configuration stated before, all background processes are terminated, and the experiment is conducted

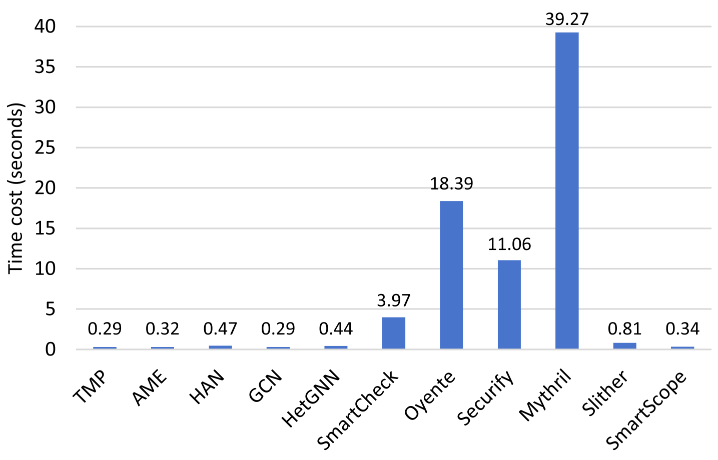


Fig 5. Time cost comparison between the selected baselines and our tool.

Table 8
Component-wise time cost of SMARTSCOPE.

	SRCG Construction	Strategy Generation	Vulnerability Detection
Time Overhead (seconds)	0.22	0.013	0.102

in a clean environment. SMARTSCOPE is run 5 times, and the average detection time per contract in our dataset is recorded.

Table 8 lists the time consumption results for the three primary components of SMARTSCOPE. Specifically, the average time consumption for detecting a smart contract is 0.22 seconds for SRCG Construction, 0.013 seconds for Strategy Generation, and 0.102 seconds for Vulnerability Detection. Overall, the average time overhead of SMARTSCOPE is 0.335 seconds, and SRCG Construction accounts for the majority of SmartScope's overhead. The time overhead is acceptable for security analysts in practice.

5. Discussion

5.1. Strengths of importance coefficients

Though seemingly simple, our method proves to be a powerful solution for detecting smart contract vulnerabilities in practice. By assigning importance coefficients, SMARTSCOPE is guided to focus on semantically meaningful subgraphs. This helps reduce noise and improve specificity, which are clear advantages over purely data-driven methods. To determine the optimal values for the importance coefficients in SMARTSCOPE, we we employ a grid search method on more than 1600 parameter combinations. Finally, experimental results demonstrate the effectiveness of our work. Furthermore, the importance coefficients are empirically adjusted rather than fixed, and can be easily adapted to new vulnerability types or domains by updating the corresponding subgraph extraction rules. In future, we plan to explore adaptive bias weighting or integrate richer domain semantics by more useful techniques, *e.g.*, LLMs and reinforcement learning, to further enhance the method's flexibility and scalability.

5.2. Scale of our smart contract dataset

While our dataset already covers a reasonable number of smart contracts collected from diverse sources, building a highly effective vulnerability detection tool often requires even larger and more representative labeled datasets. However, as shown before, existing open-source datasets contain a limited number of labeled samples, and often suffer from low label quality. To address this, we adopt a hybrid labeling strategy that combines manual inspection with LLMs to generate high-quality labeled contract samples. Prior work has shown that LLMs can be used to construct smart contract vulnerability datasets (Chen et al., 2025), which supports the reliability of our approach. In future work, as we further scale our datasets with the help of LLMs, we plan to fine-tune the models to generate more reliable Solidity code, building on insights from recent advances (Peng et al., 2025).

5.3. Toward cross-contract vulnerability detection

SMARTSCOPE currently focuses on detecting vulnerabilities within individual smart contracts, which covers a wide portion of common attack patterns. However, certain vulnerabilities arise from cross-contract interactions that fall outside the scope of our current analysis. While cross-contract detection is beyond the current scope, our method is designed with extensibility in mind, making it well-suited for future adaptation to cross-contract analysis. For example, cross-contract relations can be modeled by linking the SRCGs of different contracts through a new edge type (*i.e.*, external edges).

5.4. Extensibility for handling emerging vulnerability types

SMARTSCOPE is extensible in multiple dimensions, which positions it well to potentially support emerging vulnerability types beyond those included in the current study: (1) SMARTSCOPE can extract new types of program relations as needed by leveraging the modular and customizable Slither framework. (2) The SRCG can be extended with new node and edge types to represent emerging structural and semantic information. (3) The embedding enhancement strategy can dynamically adjust feature weights to better capture the characteristics of novel vulnerabilities. (4) SMARTSCOPE's modular pipeline allows seamless integration of advanced machine learning techniques (*e.g.*, LLMs) as they become available. In future work, we plan to validate SMARTSCOPE's adaptability by incorporating newly disclosed vulnerability types and evaluating its performance on previously unseen categories.

6. Related work

6.1. Conventional detection tools

Some work uses conventional program analysis techniques to detect specific smart contract vulnerabilities. Securify (Tsankov et al., 2018) detected specified patterns extracted from control flows of contracts. FASVERIF (Wang et al., 2023) analyzed financial security in smart contracts based on formal verification. Oyente (Luu et al., 2016) executed EVM bytecode symbolically and checked for vulnerability patterns in execution traces. Mythril (Mueller, 2017) used taint analysis and symbolic execution to find vulnerability patterns. SmartCheck (Tikhomirov et al., 2018) searched for specific patterns in the XML syntax trees of contracts. ContractFuzzer (Jiang et al., 2018) employed fuzzing technology to detect various vulnerabilities in smart contracts. Contractsentry (Wang & Zhao, 2025) detected smart contract vulnerabilities by static analysis and pattern matching. SWAT (Songsom et al., 2022) performed vulnerability detection based on the Smart Contract Weakness Classification by matching patterns in Solidity code. Compared to SmartScope, which only requires to assign the coefficients of the node importance empirically, the effectiveness of all the conventional tools heavily relies on the expert rules written into the source code, limiting their accuracy and extensibility.

6.2. Deep learning-based detection tools

Many recent attempts have been made to use deep neural networks for detecting smart contract vulnerabilities. Liu et al. proposed to fuse local pattern features about expert knowledge and global graph features of smart contract code for the detection (Liu et al., 2021a,b). The feature-level fusion does not operate directly on the graph structure, which may limit the ability to represent and learning vulnerability semantics. Zhuang et al. (2021) constructed a contract graph for a smart contract function, and then leveraged a temporal message propagation network to detect the vulnerabilities. The granularity of their graph construction can be further improved to capture more semantic behaviors of smart contracts. Meanwhile, replacing the adopted neural network with a more advanced heterogeneous graph neural network can help learn more discriminative features.

SCVHunter (Luo et al., 2024) combined heterogeneous contract semantic graphs (CSGs) and a graph neural network with two-layer attention to identify vulnerabilities in smart contracts. However, manually defining meta-paths often fail to capture the complex semantics of smart contract code. Furthermore, the lack of explicit modeling of temporal and control dependencies in CSGs hinders detection performance. MTVHunter (Sun et al., 2025a) designed a multi-teacher network integrating instruction denoising and semantic complementary to a student network for detecting smart contract vulnerabilities on bytecode. Considering that the denoising process relies on the basic control-flow graph derived from source code, the SRCG can be used to provide richer semantic and structural information, potentially enhancing the effectiveness.

Clear (Chen et al., 2024) leveraged a contrastive learning model to effectively capture fine-grained correlations among contracts, and then combined the correlations and the semantic information of smart contracts to detect vulnerabilities. Incorporating domain knowledge, as demonstrated in our work, can be a promising direction to further boost the detection capability of this technique. EFEVD (Jiang et al., 2024) identified community features from smart contracts with similar semantic and syntactic structures to detect targeted vulnerabilities. The graph structures with domain-informed biases in our work can be adopted to enhance the feature extraction. GPTScan (Sun et al., 2024) combined LLMs with static analysis to detect logic vulnerabilities in smart contracts. We plan to extend our method to detect this kind of vulnerabilitie.

7. Conclusion

In this paper, we propose and implement SmartScope, a novel approach for smart contract vulnerability detection based on heterogeneous graph embedding with local semantic enhancement. By constructing a semantically rich contract graph and incorporating importance coefficients into local structures of the graph, SmartScope effectively captures fine-grained structural and semantic information critical to vulnerability identification. Our method leverages a heterogeneous graph transformer to capture complex relations among diverse program elements and uses a lightweight MLP-based classifier for final detection. Extensive experiments demonstrate that SmartScope achieves superior performance over representative traditional and deep learning-based baselines across multiple vulnerability types, including reentrancy, timestamp dependence, and infinite loop. Furthermore, ablation studies confirm the effectiveness of each core module of our work.

CRediT authorship contribution statement

Zhaoyi Meng: Conceptualization, Methodology, Writing; **Zexin Zhang:** Data curation, Software, Visualization; **Wansen Wang:** Funding acquisition, Investigation; **Jie Cui:** Resources, Validation; **Hong Zhong:** Supervision, Project administration.

Data availability

The data that has been used is confidential.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported in part by Anhui Province Natural Science Foundation under Grant 2408085MF167, in part by National Natural Science Foundation of China under Grant 62402001, Grant U23A20308, Grant 62272002, and Grant 62102385.

References

Blockonomi (2025). Kiloex DEX experiences $7.5 million security breach due to price oracle vulnerability. https://www.binance.com/en/square/post/22943435084626. April 12, 2025.

Cai, J., Chen, J., Zhang, T., Luo, X., Sun, X., & Li, B. (2025). Detecting reentrancy vulnerabilities for solidity smart contracts with contract standards-based rules. *IEEE Transactions on Information Forensics and Security*, *20*, 3662–3676.

Chen, J., Shen, Y., Zhang, J., Li, Z., Grundy, J., Shao, Z., Wang, Y., Wang, J., Chen, T., & Zheng, Z. (2025). Forge: An llm-driven framework for large-scale smart contract vulnerability dataset construction. *arXiv preprint arXiv:2506.18795*.

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., & Chen, T. (2021). Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering*, *48*(7), 2189–2207.

Chen, Y., Sun, Z., Gong, Z., & Hao, D. (2024). Improving smart contract security with contrastive learning-based vulnerability detection. In *Proceedings of the IEEE/ACM 46th international conference on software engineering* (pp. 1–11).

Dannen, C. (2017). Solidity programming. In *Introducing ethereum and solidity: Foundations of cryptocurrency and blockchain programming for beginners* (pp. 69–88). Springer.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the north american chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)* (pp. 4171–4186).

Ding, H., Liu, Y., Piao, X., Song, H., & Ji, Z. (2025). Smartguard: An LLM-enhanced framework for smart contract vulnerability detection. *Expert Systems with Applications*, *269*, 126479.

Etherscan. https://etherscan.io/. April 22, 2025.

Feist, J., Grieco, G., & Groce, A. (2019). Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd international workshop on emerging trends in software engineering for blockchain (WETSEB)* (pp. 8–15). IEEE.

Fox, J. (2024). Smart contract security risks: Today's 10 top vulnerabilities and mitigations. https://www.cobalt.io/blog/smart-contract-security-risks. December 3, 2025.

He, Z., Liao, Z., Luo, F., Liu, D., Chen, T., & Li, Z. (2022). Tokencat: Detect flaw of authentication on ERC20 tokens. In *Icc 2022-ieee international conference on communications* (pp. 4999–5004). IEEE.

Ivanov, N., Li, C., Yan, Q., Sun, Z., Cao, Z., & Luo, X. (2023). Security threat mitigation for smart contracts: A comprehensive survey. *ACM Computing Surveys*, *55*(14s), 1–37.

Jiang, B., Liu, Y., & Chan, W. K. (2018). Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering* (pp. 259–269).

Jiang, C., Liu, X., Wang, S., Liu, J., & Zhang, Y. (2024). Efevd: Enhanced feature extraction for smart contract vulnerability detection. In *Proceedings of the thirty-third international joint conference on artificial intelligence* (pp. 4246–4254).

Lakadawala, H., Dzigbede, K., & Chen, Y. (2024). Detecting reentrancy vulnerability in smart contracts using graph convolution networks. In *2024 IEEE 21st consumer communications & networking conference (CCNC)* (pp. 188–193). IEEE.

Langley, M. (2025). $5 million stolen from 1inch due to smart contract flaw. https://dailysecurityreview.com/security-spotlight/5-million-stolen-from-1inch-due-to-smart-contract-flaw/. March 10, 2025.

Li, B., Pan, Z., & Hu, T. (2024a). Evofuzzer: An evolutionary fuzzer for detecting reentrancy vulnerability in smart contracts. *IEEE Transactions on Network Science and Engineering*, *11*(6), 5790–5802.

Li, H., Xiong, G., Hou, C., Gou, G., Chen, Z., & Li, Z. (2024b). Smart contract vulnerability detection based on AST-augmented heterogeneous graphs. In *2024 IEEE international performance, computing, and communications conference (IPCCC)* (pp. 1–10). IEEE.

Liao, Z., Nan, Y., Liang, H., Hao, S., Zhai, J., Wu, J., & Zheng, Z. (2024). Smartaxe: Detecting cross-chain vulnerabilities in bridge smart contracts via fine-grained static analysis. *Proceedings of the ACM on Software Engineering*, *1*(FSE), 249–270.

Liao, Z., Zheng, Z., Chen, X., & Nan, Y. (2022). Smartdagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 752–764).

Lin, S.-Y., Zhang, L., Li, J., Ji, L.-l., & Sun, Y. (2022). A survey of application research based on blockchain smart contract. *Wireless Networks*, *28*(2), 635–690.

Liu, Y., Xue, Y., Wu, D., Sun, Y., Li, Y., Shi, M., & Liu, Y. (2025). Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. In *Ndss*. IEEE.

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

Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., & Wang, X. (2021b). Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, *35*(2), 1296–1310.

Luo, F., Luo, R., Chen, T., Qiao, A., He, Z., Song, S., Jiang, Y., & Li, S. (2024). Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In *Proceedings of the IEEE/ACM 46th international conference on software engineering* (pp. 1–13).

Luo, R., Luo, F., Wang, B., & Chen, T. (2022). Smart contract vulnerability detection based on variant LSTM. In *Proceedings of the 2022 international conference on big data, iot, and cloud computing* (pp. 1–4).

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (pp. 254–269).

Ma, A., Wang, X., Li, J., Wang, C., Xiao, T., Liu, Y., Cheng, H., Wang, J., Li, Y., Chang, Y., Li, J., Wang, D., Jiang, Y., Su, L., Xin, G., Gu, S., Li, Z., Liu, B., Xu, D., & Ma, Q. (2023). Single-cell biological network inference using a heterogeneous graph transformer. *Nature Communications*, *14*(1), 964.

Mueller, B. (2017). A framework for bug hunting on the ethereum blockchain. https://github.com/ConsenSysDiligence/mythril. September 29, 2025.

Nguyen, H. H., Nguyen, N.-M., Doan, H.-P., Ahmadi, Z., Doan, T.-N., & Jiang, L. (2022a). Mando-guru: Vulnerability detection for smart contract source code by heterogeneous graph embeddings. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 1736–1740).

Nguyen, H. H., Nguyen, N.-M., Xie, C., Ahmadi, Z., Kudendo, D., Doan, T.-N., & Jiang, L. (2022b). Mando: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities. In *2022 IEEE 9th international conference on data science and advanced analytics (DSAA)* (pp. 1–10). IEEE.

Nguyen, H. H., Nguyen, N.-M., Xie, C., Ahmadi, Z., Kudendo, D., Doan, T.-N., & Jiang, L. (2023). Mando-hgt: Heterogeneous graph transformers for smart contract vulnerability detection. In *2023 IEEE/ACM 20th international conference on mining software repositories (MSR)* (pp. 334–346). IEEE.

Nguyen, T.-T., & Vo, H. D. (2024). Context-based statement-level vulnerability localization. *Information and Software Technology*, *169*, 107406.

Peng, Z., Yin, X., Qian, R., Lin, P., Liu, Y., Ying, C., & Luo, Y. (2025). Soleval: Benchmarking large language models for repository-level solidity code generation. *arXiv preprint arXiv:2502.18793*.

Qian, P., Liu, Z., He, Q., Zimmermann, R., & Wang, X. (2020). Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, *8*, 19685–19695.

Qian, P., Liu, Z., Yin, Y., & He, Q. (2023a). Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In *Proceedings of the ACM web conference 2023* (pp. 2220–2229).

Qian, P., Liu, Z., Yin, Y., & He, Q. (2023b). Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In *Proceedings of the ACM web conference 2023* WWW '23 (p. 2220–2229). New York, NY, USA: Association for Computing Machinery.

Shashank (2025). Web3 hackhub: Gain insights into blockchain hacks. https://solidityscan.com/web3hackhub. July 7, 2025.

Songsom, N., Werapun, W., Suaboot, J., & Rattanavipanon, N. (2022). The swc-based security analysis tool for smart contract vulnerability detection. In *2022 6th international conference on information technology (inCIT)* (pp. 74–77). IEEE.

Stephens, J., Ferles, K., Mariano, B., Lahiri, S., & Dillig, I. (2021). Smartpulse: Automated checking of temporal properties in smart contracts. In *2021 IEEE symposium on security and privacy (SP)* (pp. 555–571). IEEE.

Sun, G., Zhuang, Y., Zhang, S., Feng, X., Liu, Z., & Zhang, L. (2025a). Mtvhunter: Smart contracts vulnerability detection based on multi-teacher knowledge translation. In *Proceedings of the AAAI conference on artificial intelligence* (pp. 15169–15176). (*vol. 39*).

Sun, X., Zhou, M., Cao, S., Wu, X., Bo, L., Wu, D., Li, B., & Xiang, Y. (2025b). HgtJIT: Just-in-time vulnerability detection based on heterogeneous graph transformer. *IEEE Transactions on Dependable and Secure Computing*. https://doi.org/10.1109/TDSC.2025.3586669

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., & Liu, Y. (2024). 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).

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018). Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain* (pp. 9–16).

Torres, C. F., Iannillo, A. K., Gervais, A., & State, R. (2021). Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European symposium on security and privacy (euros&p)* (pp. 103–119). IEEE.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., & Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (pp. 67–82).

Vranken, H. (2017). Sustainability of bitcoin and blockchains. *Current Opinion in Environmental Sustainability*, *28*, 1–9.

Wang, S., & Zhao, X. (2025). Contractsentry: A static analysis tool for smart contract vulnerability detection. *Automated Software Engineering*, *32*(1), 1.

Wang, W., Huang, W., Meng, Z., Xiong, Y., Miao, F., Fang, X., Tu, C., & Ji, R. (2023). Automated inference on financial security of ethereum smart contracts. In *32nd USENIX security symposium (USENIX security 23)* (pp. 3367–3383).

Wang, W., Xia, L., Zhang, Z., & Meng, X. (2024a). Smart contract timestamp vulnerability detection based on code homogeneity. *IEICE Transactions on Information and Systems*, *107*(10), 1362–1366.

Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., & Yu, P. S. (2019). Heterogeneous graph attention network. In *The world wide web conference* (pp. 2022–2032).

Wang, Y., Zhao, X., He, L., Zhen, Z., & Chen, H. (2024b). ContractGNN: Ethereum smart contract vulnerability detection based on vulnerability sub-graphs and graph neural networks. *IEEE Transactions on Network Science and Engineering*, *11*(6), 6382–6395.

Wang, Z., Chen, J., Zheng, P., Zhang, Y., Zhang, W., & Zheng, Z. (2024c). Unity is strength: Enhancing precision in reentrancy vulnerability detection of smart contract analysis tools. *IEEE Transactions on Software Engineering*, *51*(1), 1–13.

Xu, Q., Wang, L., Sheng, W., Wang, Y., Xiao, C., Ma, C., & An, W. (2024). Heterogeneous graph transformer for multiple tiny object tracking in RGB-t videos. *IEEE Transactions on Multimedia*, *26*, 9383–9397.

Ye, G., Liu, X., Fan, S., Tan, Y., Zhou, Q., Zhou, R., & Zhou, X. (2023). Novel supply chain vulnerability detection based on heterogeneous-graph-driven hash similarity in iot. *Future Generation Computer Systems*, *148*, 201–210.

Ye, M., Nan, Y., Dai, H.-N., Yang, S., Luo, X., & Zheng, Z. (2024). Funfuzz: A function-oriented fuzzer for smart contract vulnerability detection with high effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology*, *33*(7), 1–20.

Zheng, P., Zheng, Z., & Luo, X. (2022). Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 740–751).

Zhu, H., Li, H., & Lu, G. (2025). Hsvdetector: A heterogeneous semantic graph-based method for smart contract vulnerability detection. *The Journal of Supercomputing*, *81*(4), 1–36.

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., & He, Q. (2021). Smart contract vulnerability detection using graph neural networks. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence* (pp. 3283–3290).