International IT Journal of Research (IITJR), ISSN: 3007-6706
Volume 3, Issue 3, July-September, 2025
Available online at: https://itjournal.org/index.php

Automated Smart Contract Vulnerability Auditing Using Graph Neural Networks and Symbolic

Dr. Matteo Pellegrini

Senior Lecturer, Department of Informatics, Systems and Communication, University of Milano-Bicocca, Italy

ABSTRACT

Execution
The rapid adoption of blockchain technology and decentralized applications has significantly increased the importance of smart contract security. Smart contracts are self-executing programs deployed on blockchain networks, responsible for managing digital assets, financial transactions, and decentralized operations. However, vulnerabilities in smart contract code can expose blockchain systems to severe security threats, including reentrancy attacks, integer overflow and underflow, unauthorized access, timestamp manipulation, denial-of-service attacks, and transaction-ordering exploitation. Traditional smart contract auditing techniques, including static analysis, dynamic testing, and manual code inspection, often face limitations in detecting complex vulnerabilities, particularly those involving intricate control-flow dependencies and execution paths. To address these challenges, this study proposes an automated smart contract vulnerability auditing framework that integrates Graph Neural Networks (GNNs) and Symbolic Execution to improve vulnerability detection accuracy, scalability, and interpretability.

The proposed framework represents smart contract source code and intermediate representations as graph structures, capturing relationships among functions, variables, control-flow blocks, data dependencies, and transaction states. Graph Neural Networks are employed to learn structural and semantic representations of smart contract programs, enabling the identification of suspicious execution patterns and potentially vulnerable code regions. Subsequently, symbolic execution is integrated to systematically explore feasible execution paths, analyze path constraints, and validate potential vulnerabilities through logical reasoning. The combination of learning-based detection and constraint-based verification provides a complementary approach that addresses the limitations of individual auditing techniques. The framework is designed to prioritize detected vulnerabilities according to their severity, exploitability, and potential impact, while generating detailed explanations to support security analysts and developers.

The proposed methodology can be evaluated using publicly available smart contract vulnerability datasets, benchmark contracts, and simulated attack scenarios. Performance assessment may consider precision, recall, F1-score, detection accuracy, false-positive rate, execution time, and vulnerability coverage. Comparative evaluation against conventional static analyzers, machine-learning-based classifiers, and symbolic execution tools can establish the effectiveness of the integrated approach. The framework aims to reduce manual auditing efforts, improve the reliability of vulnerability identification, and support continuous security assessment throughout the smart contract development lifecycle. Ultimately, this research contributes to the development of intelligent, scalable, and explainable blockchain security solutions, strengthening trust, transparency, and resilience in decentralized ecosystems.

Keywords: Smart Contract Security, Graph Neural Networks, Symbolic Execution, Automated Vulnerability Detection, Blockchain Auditing.

INTRODUCTION

Blockchain technology has emerged as a transformative innovation in the digital ecosystem by providing decentralized, transparent, secure, and tamper-resistant mechanisms for storing and exchanging information. One of its most significant applications is the development of smart contracts, which are self-executing programs that automatically enforce predefined agreements when specified conditions are satisfied. Smart contracts have become fundamental components of decentralized finance (DeFi), supply chain management, digital identity systems, healthcare applications, non-fungible tokens (NFTs), and decentralized autonomous organizations (DAOs). Platforms such as Ethereum have significantly expanded the adoption of smart contracts by enabling developers to create programmable blockchain-based applications. Despite these advantages, the increasing complexity and financial importance of smart contracts have introduced substantial security challenges.

Smart contract vulnerabilities represent a critical threat to blockchain infrastructure because deployed contracts often manage valuable digital assets and execute transactions without requiring centralized supervision. Unlike conventional

software applications, smart contracts deployed on public blockchains are generally difficult and costly to modify after deployment. Consequently, even a minor programming error can result in irreversible financial losses, unauthorized access, manipulation of contract execution, or disruption of decentralized services. Common vulnerabilities include reentrancy attacks, integer overflow and underflow, improper access control, unchecked external calls, transaction-ordering dependencies, denial-of-service conditions, and insecure randomness generation. Several historical blockchain security incidents have demonstrated how attackers can exploit weaknesses in contract logic to compromise digital assets and undermine user confidence.

Traditional smart contract auditing approaches primarily rely on manual code reviews, static analysis, dynamic testing, and rule-based vulnerability detection. Manual auditing can identify sophisticated logical errors but requires considerable expertise, time, and financial resources. Static analysis tools examine source code or intermediate representations without executing the contract, identifying suspicious programming patterns and potential security weaknesses. Although these tools are useful for early-stage vulnerability detection, they may generate false-positive results and struggle with complex interprocedural dependencies. Dynamic testing evaluates contract behavior during execution, but its effectiveness depends heavily on the quality and coverage of test cases. Furthermore, conventional testing techniques may fail to explore rare execution paths or complex transaction sequences that lead to security vulnerabilities.

Recent developments in artificial intelligence and machine learning have introduced new possibilities for automated smart contract security auditing. Deep learning techniques can identify complex patterns in source code and learn relationships between programming structures and known vulnerability classes. In particular, Graph Neural Networks (GNNs) have attracted significant research attention because software programs can naturally be represented as graphs. Smart contract code contains interconnected components, including functions, variables, control-flow statements, data dependencies, and function calls. Representing these components as graph structures enables GNNs to capture both local and global relationships that may indicate vulnerable programming behavior.

Graph Neural Networks operate through message-passing mechanisms that aggregate information from neighboring nodes and generate meaningful representations of graph structures. In smart contract auditing, individual nodes may represent program statements, functions, variables, or instructions, while edges describe control-flow dependencies, data-flow relationships, and function-call connections. By learning from these graph representations, GNN-based models can classify vulnerable code segments, identify suspicious interactions, and detect structural patterns associated with different vulnerability categories. However, purely learning-based approaches may face challenges when dealing with previously unseen vulnerabilities, complex execution conditions, and logically constrained attack scenarios.

Proposed Models and Methodologies

The proposed research develops an integrated automated smart contract vulnerability auditing framework using **Graph Neural Networks (GNNs) and Symbolic Execution (SE)**. The primary objective is to combine the pattern-learning capabilities of graph-based deep learning with the logical verification capabilities of symbolic program analysis. The framework is designed to identify vulnerable code structures, prioritize suspicious execution paths, validate potential security weaknesses, and generate meaningful vulnerability reports.

The proposed methodology consists of multiple stages, beginning with smart contract data collection and preprocessing, followed by graph construction, feature extraction, GNN-based vulnerability prediction, symbolic execution, vulnerability verification, and performance evaluation.

2.1 Proposed System Architecture

The proposed system follows a hybrid architecture in which Graph Neural Networks identify suspicious code patterns and symbolic execution verifies whether the predicted vulnerabilities can occur under feasible execution conditions.

The major components are:

1. Smart contract dataset collection.
2. Source code preprocessing and normalization.
3. Graph-based program representation.
4. Feature extraction and embedding generation.
5. GNN-based vulnerability classification.
6. Risk-based execution-path prioritization.
7. Symbolic execution and constraint solving.
8. Vulnerability verification and classification.
9. Automated security report generation.

Proposed workflow:
Smart Contract Dataset Collection
Solidity source code, benchmark contracts, vulnerable and secure samples

Source Code Preprocessing
Parsing, normalization, tokenization and feature extraction

Graph Construction
Control-flow, data-flow and function-call graphs

Graph Neural Network Model
Graph feature learning and vulnerability prediction

Risk-Based Path Prioritization
Selection of suspicious functions and code regions

Symbolic Execution Engine
Path exploration, symbolic inputs and constraint solving

Vulnerability Verification
Feasibility analysis and exploit-condition validation

Automated Security Audit Report
Vulnerability category, severity, affected code and evidence

2.2 Smart Contract Dataset Collection
The first stage involves collecting smart contract source code from publicly available blockchain repositories, security benchmarks, and vulnerability datasets. Solidity is considered the primary programming language because of its widespread use in Ethereum-based smart contracts.

The dataset should contain both vulnerable and secure contracts to support supervised learning and reliable model evaluation.

Potential vulnerability categories include:
- Reentrancy vulnerabilities.
- Integer overflow and underflow.
- Improper access control.
- Unchecked external calls.
- Denial-of-service vulnerabilities.
- Timestamp dependence.
- Transaction-ordering vulnerabilities.
- Uninitialized storage variables.
- Delegatecall-related vulnerabilities.

The dataset can be divided into training, validation, and testing subsets. Care should be taken to prevent duplicated contracts and closely related code versions from appearing across different subsets, as this could lead to data leakage and misleading evaluation results.

2.3 Source Code Preprocessing and Feature Extraction
Preprocessing transforms raw smart contract source code into a structured representation suitable for machine learning and symbolic analysis.
The following operations are performed:

A. Lexical Analysis: The source code is divided into tokens, including keywords, operators, identifiers, variables, and function declarations.

B. Syntax Analysis: An abstract syntax tree (AST) is generated to identify the hierarchical structure of the program.

C. Code Normalization: Variable names and irrelevant formatting differences are normalized while preserving security-relevant semantic information.

D. Dependency Extraction: Relationships among variables, functions, statements, and external calls are extracted.
E. Vulnerability Labeling: Contract samples and relevant code regions are labeled according to known vulnerability categories.
The extracted features include:

Feature category	Extracted information
Syntax features	Operators, statements, function declarations
Control-flow features	Conditional branches, loops, execution sequences
Data-flow features	Variable assignments and dependencies
Function features	Internal and external function calls
Security features	Access modifiers, external calls, arithmetic operations
Semantic features	State modifications and execution conditions

These features provide the foundation for constructing graph representations of smart contract programs.

RESULTS & ANALYSIS

The results and analysis section evaluates the proposed automated smart contract vulnerability auditing framework based on Graph Neural Networks (GNNs) and Symbolic Execution (SE). The primary objective is to examine the effectiveness of the hybrid approach in identifying security vulnerabilities, improving detection accuracy, reducing false-positive predictions, and optimizing execution-path exploration.

The proposed framework combines graph-based feature learning with symbolic verification to identify vulnerabilities that may remain undetected by conventional auditing techniques. The evaluation focuses on vulnerability classification, detection performance, execution efficiency, computational overhead, and comparative effectiveness.

Note: The numerical values presented in the following tables are illustrative experimental results designed to demonstrate the proposed evaluation methodology. They should be replaced with actual measurements after implementation and testing.

3.1 Experimental Evaluation Setup

The proposed framework is evaluated using smart contract datasets containing vulnerable and non-vulnerable Solidity programs. The dataset includes different vulnerability categories, such as reentrancy, integer overflow, access-control weaknesses, unchecked external calls, and denial-of-service vulnerabilities.

The experimental environment consists of the following components:

Parameter	Experimental configuration
Programming language	Python
Smart contract language	Solidity
Deep learning framework	PyTorch
Graph learning library	PyTorch Geometric
GNN architecture	Graph Attention Network (GAT)
Symbolic execution	Mythril / equivalent engine
Constraint solver	Z3
Dataset	Vulnerable and secure smart contracts
Training ratio	70%
Validation ratio	15%
Testing ratio	15%
Evaluation method	Classification and execution analysis

The evaluation compares four different approaches:

1. Traditional static analysis.
2. Machine-learning-based vulnerability detection.
3. Symbolic execution-based auditing.
4. Proposed hybrid GNN and Symbolic Execution framework.

3.2 Overall Vulnerability Detection Performance

The first experiment evaluates the overall vulnerability detection performance of different auditing approaches. Accuracy, precision, recall, and F1-score are considered the primary evaluation metrics.

Table 1. Overall performance comparison

Method	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Static Analysis	85.2	82.4	80.6	81.5
Machine Learning	88.7	87.1	85.4	86.2
Symbolic Execution	90.1	89.3	88.2	88.7
Proposed Hybrid GNN-SE	**95.6**	**94.8**	**93.9**	**94.3**

The illustrative results indicate that the proposed hybrid framework achieves the highest performance across all four evaluation metrics. Its accuracy of 95.6% suggests improved classification capability compared with traditional static analysis, machine learning, and standalone symbolic execution.

The improvement can be attributed to the complementary characteristics of the integrated architecture. GNNs learn structural and semantic relationships among program components, while symbolic execution validates suspicious execution paths using logical constraints.

The results also indicate that the hybrid approach may reduce the limitations associated with relying exclusively on statistical predictions or rule-based analysis.

3.3 Vulnerability-Wise Detection Analysis

The second experiment examines the detection performance of the proposed framework across different categories of smart contract vulnerabilities.

Table 2. Vulnerability-wise detection performance

Vulnerability category	Detection rate (%)	Precision (%)
Reentrancy	97.2	96.1
Integer Overflow/Underflow	95.4	94.7
Access Control	94.8	93.5
Unchecked External Calls	93.6	92.8
Denial of Service	92.7	91.9
Timestamp Dependence	94.1	93.2
Delegatecall Risks	91.8	90.6

The illustrative findings demonstrate strong detection performance across multiple vulnerability categories.
Reentrancy vulnerabilities achieve the highest detection rate because their identification can benefit from analyzing function-call relationships, external calls, and state-update dependencies.

Integer overflow and underflow vulnerabilities are identified through arithmetic-operation analysis and symbolic boundary checking. Access-control vulnerabilities are evaluated through function modifiers, caller permissions, and authorization conditions.

The comparatively lower detection rate for delegatecall-related vulnerabilities highlights the complexity of analyzing dynamic execution targets and cross-contract storage interactions.

Overall, the results suggest that combining graph-based learning and symbolic verification can support the identification of multiple vulnerability classes within a unified auditing framework.

COMPARATIVE ANALYSIS

The comparative analysis evaluates the proposed **Automated Smart Contract Vulnerability Auditing Using Graph Neural Networks and Symbolic Execution** against conventional static analysis, machine-learning-based detection, deep learning-based analysis, and standalone symbolic execution techniques.

The comparison focuses on vulnerability detection accuracy, precision, recall, false-positive rate, execution time, vulnerability coverage, scalability, and verification capability.

International IT Journal of Research (IITJR), ISSN: 3007-6706
Volume 3, Issue 3, July-September, 2025
Available online at: https://itjournal.org/index.php

4.1 Comparative Performance Analysis

Table 1. Comparison of different smart contract auditing approaches

Evaluation parameters	Static Analysis	Machine Learning	Symbolic Execution	Proposed GNN-SE
Detection accuracy (%)	85.2	88.7	90.1	**95.6**
Precision (%)	82.4	87.1	89.3	**94.8**
Recall (%)	80.6	85.4	88.2	**93.9**
F1-score (%)	81.5	86.2	88.7	**94.3**
False-positive rate (%)	14.8	11.6	8.9	**4.3**
Execution time (sec)	8.5	5.2	28.6	12.4
Path verification	Limited	No	Yes	**Yes**
Graph-based learning	No	Limited	No	**Yes**

4.2 Vulnerability Detection Capability

Table 2. Comparative analysis of vulnerability detection capabilities

Vulnerability category	Static Analysis	Machine Learning	Symbolic Execution	Proposed GNN-SE
Reentrancy	Moderate	High	High	Very High
Integer overflow	High	High	Very High	Very High
Access control	Moderate	High	High	Very High
Unchecked external calls	High	Moderate	High	Very High
Denial of service	Moderate	Moderate	High	Very High
Timestamp dependence	Moderate	Moderate	High	Very High
Delegatecall risks	Moderate	High	High	Very High

Qualitative assessment based on the intended capabilities of each approach, not independently measured benchmark results.

4.3 Computational Efficiency Comparison

Table 3. Computational efficiency and resource utilization

Parameters	Static Analysis	Machine Learning	Symbolic Execution	Proposed GNN-SE
Processing speed	High	Very High	Low	High
Memory requirements	Low	Moderate	High	Moderate–High
Computational complexity	Low–Moderate	Moderate	High	Moderate–High
Execution-path exploration	Limited	No direct exploration	Extensive	Risk-guided
Large-contract handling	Good	Good	Challenging	Potentially improved
Scalability	High	High	Limited by path explosion	Dependent on graph and path size

4.4 Feature-Based Comparative Analysis

Table 4. Comparison of methodological features

Features	Traditional Auditing	ML-Based Auditing	SE-Based Auditing	Proposed Hybrid Framework
Automated vulnerability detection	Yes	Yes	Yes	Yes
Structural code analysis	Yes	Limited	Yes	Yes
Graph representation	No	Optional	No	Yes
Dependency analysis	Rule-based	Learned patterns	Execution-based	Integrated
Logical constraint solving	No	No	Yes	Yes
Vulnerability classification	Rule-based	Yes	Property-based	Yes
Execution-path prioritization	Limited	Prediction-based	Search-based	GNN-guided
Vulnerability verification	Limited	Limited	Yes	Yes
Explainable findings	Rule-based	Model-dependent	Execution traces	Graph and execution evidence

LIMITATIONS & DRAWBACKS

Although the proposed framework, **Automated Smart Contract Vulnerability Auditing Using Graph Neural Networks and Symbolic Execution**, offers a promising approach to improving blockchain security, several technical, computational, and practical limitations must be considered. The integration of Graph Neural Networks (GNNs) and Symbolic Execution (SE) improves the potential for automated vulnerability identification and verification; however, the framework is not completely free from challenges.

The major limitations and drawbacks are discussed below.

5.1 High Computational Complexity

The integration of graph-based learning and symbolic execution requires considerable computational resources. Graph Neural Networks perform multiple message-passing operations to learn relationships between program components, while symbolic execution explores different execution paths and generates logical constraints.

When smart contracts contain numerous functions, conditional statements, loops, and external calls, computational requirements may increase significantly. Consequently, analyzing large and complex smart contracts can require substantial processing time and memory.

5.2 Path Explosion Problem

Symbolic execution faces the well-known path explosion problem. As the number of conditional branches and execution possibilities increases, the number of potential execution paths can grow exponentially.

Although GNN-based prioritization can reduce unnecessary path exploration, it cannot guarantee that every vulnerable execution path will be examined. Some vulnerabilities may remain undetected if their associated paths receive low prioritization scores.

5.3 Dependence on Dataset Quality

The performance of GNN-based vulnerability detection depends heavily on the quality, diversity, and reliability of the training dataset.

Incomplete, imbalanced, outdated, or incorrectly labeled smart contract datasets may negatively affect model performance. Furthermore, models trained on specific vulnerability patterns may struggle to identify previously unseen vulnerabilities or contracts developed using different programming practices.

5.4 False-Positive and False-Negative Predictions

Although symbolic verification can reduce false-positive predictions, the proposed framework cannot eliminate classification errors completely.

False positives may occur when the model identifies suspicious code patterns that do not represent actual security vulnerabilities. False negatives may arise when vulnerabilities are hidden within complex dependencies, unusual execution sequences, or previously unseen programming patterns.

The reliability of the framework therefore depends on accurate vulnerability specifications, graph representations, and execution constraints.

5.5 Graph Construction Challenges

The effectiveness of GNN-based analysis depends on the quality of the graph representation generated from smart contract source code.

Incorrectly constructed control-flow graphs, data-flow graphs, or function-call graphs may result in the loss of important semantic relationships. Complex inheritance structures, dynamic function calls, external contract interactions, and assembly-level instructions can further complicate graph construction.

Consequently, graph-generation errors may directly influence vulnerability detection performance.

CONCLUSION

The increasing adoption of blockchain technology and decentralized applications has made smart contract security a critical concern in modern digital infrastructure. Smart contracts manage valuable digital assets and execute transactions automatically; therefore, even minor programming vulnerabilities can result in significant financial losses, unauthorized access, and disruption of decentralized services. Traditional smart contract auditing techniques, including manual inspection, static analysis, dynamic testing, and conventional symbolic execution, provide valuable security assessment capabilities but face limitations in detecting complex vulnerabilities, reducing false positives, and efficiently exploring extensive execution paths.

This research presents an automated smart contract vulnerability auditing framework based on the integration of **Graph Neural Networks (GNNs) and Symbolic Execution**. The proposed approach combines graph-based structural learning with constraint-based program verification to improve the identification and validation of security weaknesses in smart contract applications. Smart contract source code is transformed into graph representations incorporating control-flow relationships, data dependencies, and function-call structures. The GNN model analyzes these representations to identify suspicious code patterns and prioritize potentially vulnerable program regions. Subsequently, symbolic execution examines relevant execution paths, generates path constraints, and verifies whether identified vulnerabilities can occur under feasible execution conditions.

The proposed methodology addresses several important smart contract security challenges, including reentrancy attacks, integer overflow and underflow, improper access control, unchecked external calls, denial-of-service vulnerabilities, timestamp dependence, and delegatecall-related risks. The integration of learning-based prediction and symbolic verification offers a more comprehensive auditing strategy than relying exclusively on individual detection techniques.

The illustrative experimental evaluation presented in this research demonstrates the potential advantages of the hybrid architecture. The proposed framework achieves an indicative detection accuracy of 95.6%, precision of 94.8%, recall of 93.9%, and F1-score of 94.3%, with a false-positive rate of 4.3%. Furthermore, GNN-guided execution-path prioritization demonstrates the potential to reduce unnecessary symbolic exploration and improve computational efficiency. These numerical findings are illustrative and require confirmation through actual implementation, independent benchmark testing, and reproducible experiments.

The research also highlights the importance of addressing challenges associated with computational complexity, path explosion, dataset dependency, graph construction, cross-contract interactions, and the detection of previously unseen vulnerabilities. Future developments can incorporate graph transformers, reinforcement learning, self-supervised learning, advanced constraint-solving techniques, and explainable artificial intelligence to improve detection reliability and scalability.

In conclusion, the integration of Graph Neural Networks and Symbolic Execution represents a promising advancement in automated smart contract security auditing. By combining intelligent vulnerability prediction, structural dependency analysis, and logical execution verification, the proposed framework establishes a foundation for more accurate, efficient, and explainable blockchain security assessment. Its further development and real-world validation can contribute significantly to strengthening decentralized applications, protecting digital assets, and improving trust in blockchain-based ecosystems.

REFERENCES

1. Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust* (pp. 164–186). Springer. https://doi.org/10.1007/978-3-662-54455-6_8
2. Bragagnolo, S., Rocha, H., Denker, M., & Ducasse, S. (2018). SmartInspect: Solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE.
3. Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (pp. 209–224).
4. Chen, T., Li, X., Luo, X., & Zhang, X. (2017). Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 442–446). IEEE. https://doi.org/10.1109/SANER.2017.7884650
5. 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. https://doi.org/10.1109/WETSEB.2019.00008
6. Grieco, G., Song, W., Cygan, A., Feist, J., & Groce, A. (2020). Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 557–560). ACM. https://doi.org/10.1145/3395363.3404366
7. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., & Zohar, Y. (2017). Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages, 2*(POPL), Article 48. https://doi.org/10.1145/3158136
8. Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). ZEUS: Analyzing safety of smart contracts. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2018.23082
9. King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM, 19*(7), 385–394. https://doi.org/10.1145/360248.360252
10. Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1609.02907
11. 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). ACM. https://doi.org/10.1145/2976749.2978309
12. Meng, Z., Zhang, Z., Wang, W., Cui, J., & Zhong, H. (2026). SmartScope: Smart contract vulnerability detection via heterogeneous graph embedding with local semantic enhancement. *Expert Systems with Applications*. https://doi.org/10.1016/j.eswa.2025.129857
13. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., & Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (pp. 653–663). ACM. https://doi.org/10.1145/3274694.3274743
14. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, 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). ACM. https://doi.org/10.1145/3243734.3243780
15. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1710.10903
16. Wang, W., Song, J., Xu, G., Li, Y., Wang, H., & Su, C. (2019). ContractWard: Automated vulnerability detection models for Ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*.
17. Wu, G., Wang, H., Lai, X., Wang, M., He, D., & Chan, S. (2024). A comprehensive survey of smart contract security: State of the art and research directions. *Journal of Network and Computer Applications, 226*, Article 103882.
18. Xu, J., Wang, T., Lv, M., Chen, T., Zhu, T., et al. (2024). MVD-HG: Multigranularity smart contract vulnerability detection method based on heterogeneous graphs. *Cybersecurity, 7*, Article 55.
19. Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., & He, Q. (2020). Smart contract vulnerability detection using graph neural network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 3283–3290). https://doi.org/10.24963/ijcai.2020/454
20. Zhou, H., Milani Fard, A., & Makanju, A. (2022). The state of Ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy, 2*(2), 358–378. https://doi.org/10.3390/jcp2020019