

Pre-Model Representation Failures in GNN-Based Smart Contract Vulnerability Detection

Birindwa Prisca Hondi ✉ 

Carnegie Mellon University Africa, Kigali, Rwanda

Chinonso Philip Nwishienyi ✉ 

Carnegie Mellon University Africa, Kigali, Rwanda

Charity Wanja Mwaura ✉ 

Carnegie Mellon University Africa, Kigali, Rwanda

Alia Teto ✉ 

Carnegie Mellon University Africa, Kigali, Rwanda

Jema David Ndibwile ✉ 

Carnegie Mellon University Africa, Kigali, Rwanda

Abstract

This paper is a failure analysis of the representation layer underlying GNN-based smart contract vulnerability detectors. These systems convert source code into graphs before any learning takes place; if the graph fails to capture the code’s semantics, no model improvement can compensate.

We investigate GNNSCVULDETECTOR and identify four failures. First, structurally different contracts produce byte-for-byte identical graphs, constituting a concrete evasion attack. Second, graph construction is governed by a hardcoded 47-entry variable whitelist (including one duplicate entry), which constrains what the extractor can recognise. As a consequence, identical vulnerabilities with different variable names produce inconsistent graphs, graph quality degrades as naming diverges from the whitelist, and when no entry matches the pipeline produces structural output not grounded in source variables. Third, the `C` node (the graph element representing the external caller that triggers a reentrancy attack) is absent from even the most canonical vulnerable contract in the literature. Fourth, a controlled experiment confirms this as a direct misclassification: a fully exploitable contract is labelled *safe* because the $C \rightarrow W$ edge is never constructed.

All four failures are demonstrated experimentally. Current accuracy figures in the literature are measured under conditions that do not expose these failures. We demonstrate one confirmed case of misclassification caused directly by a representation-layer failure; the prevalence of such failures in real-world contract populations remains an open empirical question.

2012 ACM Subject Classification Security and privacy → Systems security; Security and privacy → Software and application security; Computing methodologies → Machine learning

Keywords and phrases Smart contract security, graph neural networks, vulnerability detection, DeFi, reentrancy, feature extraction

Digital Object Identifier 10.4230/LIPIcs.AFT.2026.33

1 Introduction

This paper is a failure analysis. Every experiment targets the representation layer, the point at which source code is converted into a graph, to identify failures that occur *before* the model runs and that model improvement alone cannot resolve.

Decentralized Finance (DeFi) has grown into a complex financial ecosystem built on smart contracts. These contracts handle real assets and when they contain vulnerabilities, the losses are immediate and irreversible. This has motivated the development of automated detection systems, with Graph Neural Networks (GNNs) emerging as a popular approach. Systems such as DeFiGuard [5], DeFiTail [6], GANS-MARL [1] and related systems [7, 4] report detection accuracies between 88.9% and 98.39%.



© Birindwa Prisca Hondi, Chinonso Philip Nwishienyi, Charity Wanja Mwaura, Alia Teto, and Jema David Ndibwile;

licensed under Creative Commons License CC-BY 4.0

8th Conference on Advances in Financial Technologies (AFT 2026).

Editors: Aggelos Kiayias and Maria Kyropoulou; Article No. 33; pp. 33:1–33:12



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These numbers look promising. But they all depend on something that has not been adequately tested: the graph representation itself. Every GNN-based detector works in two steps. First, source code is converted into a graph. Second, the neural network analyses that graph. A GNN trained on millions of contracts cannot detect a vulnerability whose defining features were silently discarded during graph construction. No architectural improvement, no additional training data and no fine-tuning can compensate for information that was never extracted. This paper identifies a class of failures that model improvement cannot address.

We investigated this through four experiments on GNNSCVULDETECTOR [8]. Our findings are:

1. **Structural invariance as an evasion attack.** Contracts with different names, different functions, and added dead code produce byte-for-byte identical graphs. This is a concrete evasion attack: an attacker who renames state variables bypasses detection with no knowledge of the model’s training data or architecture.
2. **A hardcoded 47-entry whitelist.** We designed four contracts with identical reentrancy vulnerabilities differing only in variable naming. Identical vulnerabilities produce inconsistent graph encodings, graph quality degrades progressively as naming deviates from the whitelist, and when no entry matches the pipeline produces structural output not grounded in source variables.
3. **Missing caller context: an extraction inconsistency.** The `C` node (the graph element representing the external caller that triggers a reentrancy attack) is generated inconsistently: present for some contracts in the training data but absent for the most well-known vulnerable contract in the literature, with no signal from the pipeline that anything is missing.
4. **Confirmed pre-model failure.** A controlled experiment on a purpose-built minimal contract shows the model outputs label 0 (*safe*) for a fully exploitable contract due to the absent $C \rightarrow W$ edge.

These findings show that the pipeline fails at the representation layer before the model runs. The accuracy figures reported in the literature reflect performance under conditions that do not expose these failures.

2 Threat Model

We consider an attacker whose goal is to deploy a reentrancy-vulnerable smart contract that evades GNN-based detection. The attacker has black-box access to the detection pipeline: they know the system uses a GNN-based classifier but have no knowledge of the model’s weights, training data, or internal architecture. The attacker’s only capability is the ability to modify the source code before deployment, specifically by renaming state variables, functions, or the contract itself, and by inserting semantically inert dead code. The reentrancy vulnerability itself, the external-call-before-state-update pattern, is preserved intact.

Under this threat model, our experiments show that evasion is trivially achievable: Finding 1 demonstrates that surface-level renaming alone produces byte-for-byte identical graphs, and Finding 2 shows that using any state variable name outside the 47-entry whitelist produces a degenerate graph regardless of the vulnerability present. Neither capability requires knowledge of the model. The attacker needs only a text editor.

3 Background and Related Work

3.1 GNN-Based Smart Contract Detection

GNN-based detectors convert smart contract source code into a graph and train a neural network to recognise vulnerability patterns. Systems such as DeFiGuard [5], DeFiTail [6], and GANS-MARL [1] report accuracies between 88.9% and 98.39% on reentrancy and related exploits. GNNSCVULDETECTOR [8], introduced at IJCAI 2020, converts Solidity source code into a graph and classifies contracts using a GNN. This is the system we test.

3.2 What Has Been Tested and What Has Not

Existing evaluations test model accuracy on held-out contract sets. GANS-MARL [1] also tests robustness against obfuscated inputs, finding accuracy drops from 93.8% to 88.3%. However, all of these evaluations assume the graph representation is correct. None examines whether the representation preserves semantic information under modification. This paper fills that gap.

3.3 Adversarial Evasion Research

Prior adversarial work focuses on bypassing the model through crafted inputs, achieving evasion rates of 94–97% [3, 2]. This paper examines a more fundamental question: whether the representation given to the model is faithful in the first place.

4 Experimental Setup

4.1 Environment

All experiments ran on a Kali Linux machine (x86_64). GNNSCVulDetector was originally built against an older TensorFlow/Keras stack, so we reproduced that environment exactly, Python 3.7.17, TensorFlow 1.14.0, Keras 2.2.4, scikit-learn 0.20.2 and protobuf 3.20.3, inside a dedicated virtual environment to avoid dependency conflicts with newer libraries. Because GNNSCVULDETECTOR does not execute correctly under modern TensorFlow releases, reproducing the original software environment was necessary to ensure that observed failures originated from the published implementation rather than from dependency incompatibilities introduced by version drift.

4.2 Model and Pipeline

GNNSCVULDETECTOR [8] works in three steps:

1. `AutoExtractGraph.py` reads the Solidity source code and creates a graph of nodes and edges.
2. `graph2vec.py` converts the graph into numerical vectors.
3. `GNNSCModel.py` trains a GNN classifier on these vectors.

Concretely, a graph here is a structured summary of the contract’s logic: each node represents a code element such as a state variable, a function, or an external call, and each edge represents a relationship between them, for example, “this external call happens before this variable is updated.” This is conceptually similar to a flowchart of the contract’s behaviour, except built automatically from the source code rather than drawn by hand. The GNN in step 3 never sees the original Solidity code; it only ever sees this graph. If the graph omits or distorts a relationship, the model has no way to recover the missing information, regardless of how well it is trained.

■ **Table 1** Baseline performance on reentrancy detection (threshold = 0.45). Validation $F_1 = 0.848$ and $AUC = 0.896$ represent the performance ceiling when the pipeline operates as designed. False Positive Rate (FPR) is the share of safe contracts incorrectly flagged as vulnerable; AUC (area under ROC curve) summarises the model’s ability to distinguish vulnerable from safe contracts across all thresholds, with 1.0 being perfect.

Metric	Training Set	Validation Set
F1-Score	0.933	0.848
Recall	0.926	0.865
Precision	0.940	0.805
False Positive Rate	0.028	0.109
Accuracy	0.955	0.882
AUC	–	0.896

For Experiments 1 and 2, extracted graph files were compared directly at the node, edge, node-feature, and edge-feature levels. Byte-for-byte identity refers to exact equality of the serialised graph representations produced by `AutoExtractGraph.py`. The representation is created in step 1. This is where our investigation focuses.

4.3 Dataset

The dataset contains 6,200 labelled reentrancy contract graphs: 4,948 for training and 1,252 for validation. Source contracts and pre-extracted graph data are provided with the GNNSCVULDETECTOR repository [8].

5 Baseline Performance

Table 1 reports model performance when test contracts come from the same distribution as the training data, that is, contracts using variable names the extractor recognises and vulnerability patterns the graph schema can represent. These figures represent the upper bound of performance when the pipeline operates as designed; Sections 6–9 test what happens when contracts fall outside these conditions. The threshold of 0.45 means the model classifies a contract as vulnerable whenever its output score exceeds 0.45; this value was set by the original authors and is used unchanged throughout our evaluation.

Each experiment that follows tests what happens when contracts fall outside this distribution.

6 Experiment 1: Structural Invariance as an Evasion Attack

6.1 Setup

We took `simple_dao.sol`, a well-known reentrancy-vulnerable contract, and created a modified version with the following changes:

- Contract renamed: `SimpleDAO` → `VaultManager`
- Functions renamed: `donate` → `deposit`, `withdraw` → `requestFunds`, `queryCredit` → `checkBalance`
- State variable renamed: `credit` → `balances`
- Dead code inserted: internal function `_processRequest` that performs no meaningful computation

The reentrancy vulnerability was kept intact. The external call still happens before the state update.

6.2 Original Contract

Listing 1 shows the original vulnerable contract.

■ **Listing 1** simple_dao.sol: original vulnerable contract.

```

1  pragma solidity ^0.4.24;
2  contract SimpleDAO {
3      mapping (address => uint) public credit;
4
5      function donate(address to) payable public {
6          credit[to] += msg.value;
7      }
8
9      function withdraw(uint amount) public {
10         if (credit[msg.sender] < amount) { throw; }
11         require(msg.sender.call.value(amount)()); // call FIRST
12         credit[msg.sender] -= amount;             // state update AFTER
13     }
14
15     function queryCredit(address to) view public
16         returns (uint) {
17         return credit[to];
18     }
19 }

```

6.3 Modified Contract

Listing 2 shows the modified version.

■ **Listing 2** simple_dao_modified.sol: renamed and restructured, vulnerability preserved.

```

1  pragma solidity ^0.4.24;
2  contract VaultManager {
3      mapping (address => uint) public balances;
4
5      function deposit(address to) payable public {
6          balances[to] += msg.value;
7      }
8
9      function _processRequest(uint amount) internal
10         returns (bool) {
11         return amount > 0; // dead code
12     }
13
14     function requestFunds(uint amount) public {
15         if (balances[msg.sender] < amount) { throw; }
16         bool ok = _processRequest(amount);
17         require(msg.sender.call.value(amount)()); // preserved
18         balances[msg.sender] -= amount;
19     }
20
21     function checkBalance(address to) view public
22         returns (uint) {
23         return balances[to];
24     }
25 }

```

6.4 Results

Both contracts were passed through `AutoExtractGraph.py`. Table 2 shows the output.

■ **Table 2** Graph extraction output: original vs. modified. All features are identical across both contracts.

Feature	Original	Modified	Match?
Node count	5	5	✓
Edge count	3	3	✓
Node features	Identical	Identical	✓
Edge features	Identical	Identical	✓
C node present	No	No	✓
<i>Graphs are byte-for-byte identical</i>			

► **Finding 1 (Evasion Attack).** The graph extractor is invariant to contract renaming, variable renaming, function renaming, and dead code insertion. The two contracts, clearly distinct programs, are assigned byte-for-byte identical graph representations. This is not a theoretical edge case: it is a concrete evasion attack. An attacker who renames state variables bypasses detection with no knowledge of the model’s training data or architecture. The rename alone is sufficient.

7 Experiment 2: The Variable Name Whitelist

7.1 How the Extractor Constructs the Graph

To understand why Experiment 1 produced identical graphs, we examined the source code of `AutoExtractGraph.py`. The tool does not perform general analysis of the Solidity code. Instead, it looks for specific variable names using a hardcoded list, shown in Listing 3:

■ **Listing 3** Partial whitelist from `AutoExtractGraph.py` (47 entries total, including one duplicate).

```

1 var_list = [
2     'balances[msg.sender]',
3     'credit[msg.sender]',
4     'credit[to]',
5     'accountBalances[msg.sender]',
6     'userBalance[msg.sender]',
7     'Bal[msg.sender]',
8     'payments[msg.sender]',
9     'rewardsForA[recipient]',
10    '# ... 47 entries total'
11    'msg.sender'
12 ]

```

If a match is found, a node is created in the graph. If no match is found, no node is created. The duplicate entry is `participated[msg.sender]`, which appears at positions 2 and 23 of the list; its presence does not extend coverage but does confirm the list was assembled manually rather than generated systematically.

■ **Table 3** Graph output across four identical vulnerabilities with different variable naming. Three distinct failure modes are evident: inconsistency (A vs. B), degradation (A vs. C), and fabrication (D).

Contract	Variable	Nodes	Edges	Observation
A	<code>balances[msg.sender]</code> (exact match)	5	3	Normal graph
B	<code>userDeposits[msg.sender]</code> (substring)	5	3	Different edge type
C	<code>ledger[owner]</code> (no <code>msg.sender</code>)	3	1	Degraded graph
D	<code>ledger[depositor]</code> (zero matches)	3	1	Fabricated S node

7.2 Why Experiment 1 Produced Identical Graphs

Both `credit[msg.sender]` and `balances[msg.sender]` appear on the whitelist. Renaming the variable from `credit` to `balances` moved it to another recognised entry. Both contracts produced the same graph not because the tool understood they share the same vulnerability, but because both names happen to be on the list.

7.3 Experimental Proof of Whitelist Failure

To empirically demonstrate the consequences of the whitelist mechanism, we designed four contracts implementing identical reentrancy vulnerabilities that differ only in the naming of the state variable. All four share the external-call-before-state-update pattern. The four names were chosen to span the whitelist boundary: Contract A uses a name that matches the whitelist exactly, Contract B uses a name that partially overlaps a whitelist entry, Contract C uses a name with no `msg.sender` component and Contract D uses a name that matches no whitelist entry at all.

These results establish three empirical claims. Contracts A and B implement the same vulnerability yet their graphs carry different edge types, so the model receives structurally contradictory encodings for the same attack pattern based solely on variable naming. Moving further from the whitelist, node count drops from 5 to 3 and edge count from 3 to 1 between Contract A and Contract C, with no indication from the pipeline that information has been lost. At the extreme, Contract D has zero whitelist matches yet the extractor still produces 3 nodes and 1 edge: the S node it emits does not correspond to any state variable in the source; it is generated by a separate internal code path triggered by the `.call.value()` pattern, producing structural output not grounded in source variables.

► **Finding 2 (Whitelist Failure).** The graph extractor relies on a hardcoded 47-entry variable name whitelist (including one duplicate). Any contract whose state variable name falls outside this list produces a degraded or fabricated graph with no warning from the pipeline. Three failure modes are demonstrated: identical vulnerabilities produce structurally inconsistent graphs when variable names differ (Contracts A and B); graph quality degrades progressively as naming deviates from the whitelist (Contract C); and when no whitelist entry matches, the pipeline fabricates structural output not grounded in any source variable (Contract D). This is a pre-model failure – evasion occurs at the feature extraction stage before any machine learning inference takes place.

8 Experiment 3: The Missing Caller Node – An Extraction Inconsistency

8.1 What the C Node Represents

In GNNSCVULDETECTOR’s graph schema, the **C** node represents the caller function context: the external contract that initiates the call into the vulnerable function. For reentrancy attacks, this is not optional context; it is the attack’s active agent. A reentrancy attack is defined by an external contract calling the vulnerable function, which then makes an external call back before updating its state. Without a **C** node, the graph cannot represent the attack mechanism at all.

The absence of the **C** node is therefore not a minor limitation. Table 4 shows that `EtherStore.sol` in the training data does receive a $C \rightarrow W$ edge, which confirms the schema is capable of encoding the caller relationship under some conditions. The failure is one of consistency: the extractor generates this edge for some contracts and silently omits it for others implementing the same vulnerability, with no indication from the pipeline that the representation is incomplete. This is a conceptual failure in the extraction logic, not a strict impossibility in the schema.

8.2 Results

During graph extraction on both `simple_dao.sol` and its modified variant, the extractor emits:

```
"There is no C node"
```

This occurs for the most well-known reentrancy-vulnerable contract in the literature.

► **Finding 3 (Extraction Inconsistency).** Earlier versions of this work described this issue as a structural inability to encode caller context. Our experiments instead show that the schema can encode this relationship, but the extraction logic fails to do so consistently. The $C \rightarrow W$ edge is generated inconsistently across contracts implementing the same vulnerability. `EtherStore.sol` in the training data carries this edge; `simple_dao.sol` and its modified variant do not, despite sharing the same reentrancy pattern. The extractor is therefore not structurally incapable of encoding the caller relationship, but it fails to do so reliably. The consequence is the same: the model receives graphs that omit the cross-contract call dynamic that defines reentrancy for some contracts, with no signal that anything is missing. The detector classifies based on incomplete and inconsistently constructed representations of the attack it is trained to find.

9 Experiment 4: Controlled Confirmation via `userFunds.sol`

This is the central empirical result of this paper.

9.1 Motivation

Finding 3 establishes that the **C** node is absent for `simple_dao.sol`. To confirm that this absence directly causes a misclassification, we constructed a minimal purpose-built contract (`userFunds.sol`). We used `balances[msg.sender]` – a variable name that is on the whitelist – so that any misclassification could not be attributed to the whitelist failure described in Finding 2. This isolates a single question: whether the missing $C \rightarrow W$ edge alone is

■ **Table 4** Graph structure comparison: `EtherStore.sol` vs. `userFunds.sol`. The absent $C \rightarrow W$ edge in `userFunds.sol` directly produces the misclassification.

Property	<code>EtherStore.sol</code>	<code>userFunds.sol</code>
Ground truth label	1 (vulnerable)	1 (vulnerable)
C node present	Yes	Yes (isolated)
$C \rightarrow W$ edge present	Yes	No
Graph encodes reentrancy	Yes	No
Model prediction	Vulnerable ✓	Safe ×

sufficient to cause the model to mislabel an exploitable contract as safe. `userFunds.sol` was intentionally designed to isolate the caller-edge failure while avoiding whitelist effects, by using `balances[msg.sender]`, a variable name confirmed to be accepted by the extractor. Listing 4 shows the contract.

■ **Listing 4** `userFunds.sol`: minimal controlled reentrancy contract.

```

1  pragma solidity ^0.4.18;
2  contract UserFunds {
3      mapping (address => uint256) public balances;
4
5      function deposit() public payable {
6          balances[msg.sender] += msg.value;
7      }
8
9      function withdrawFunds(uint256 _weiToWithdraw) public {
10         require(balances[msg.sender] >= _weiToWithdraw);
11         // VULNERABILITY: external call before state update
12         msg.sender.call.value(_weiToWithdraw)();
13         balances[msg.sender] -= _weiToWithdraw;
14     }
15 }
```

9.2 Graph Output and Comparison

The `C0` node was created but carries a `NULL` connection field and appears in no edge; it is completely isolated. No $C \rightarrow W$ edge was generated. An isolated node contributes nothing to the GNN’s message-passing computation: because graph neural networks learn by aggregating information along edges, a node with no edges receives no neighbourhood signal and passes none. The `C0` node exists in the graph data structure but is invisible to the model’s inference process.

`EtherStore.sol` from the training data carries edge `[1,7,2]` encoding the $C \rightarrow W$ relationship. `userFunds.sol`, despite implementing the same vulnerability, produces no such edge (Table 4).

9.3 Model Inference Results

► **Finding 4.** The model achieves a best validation F_1 of 0.8511 on the training distribution yet outputs label 0 (*safe*) for `userFunds.sol`, a contract that is fully exploitable. The failure is located entirely in the graph construction layer, prior to any model inference. The

■ **Table 5** GNNSCModel.py inference results, full 250-epoch run.

Metric	Value	Notes
Total epochs run	250	Full run completed
Best training F1 epoch	242	Train F1 = 0.9340
Best validation F1	0.8511	Epoch 218
Final validation F1	0.8292	Epoch 250
Final validation accuracy	0.8786	Epoch 250
Final validation recall	0.8601	Epoch 250
Final validation FPR	0.1118	Epoch 250
Predicted label (userFunds)	0 (safe)	Pre-model failure confirmed

absent $C \rightarrow W$ edge means the GNN receives no structural signal encoding the reentrancy relationship. The pipeline produces a graph, the model produces a label, and the label is wrong, not because the model erred, but because the information required to make the correct prediction was never provided to it.

10 Discussion

10.1 Summary of Findings

Our four experiments confirm that representation-layer failures are systematic and occur upstream of the model. Because failures occur before the model runs, downstream improvements cannot compensate.

- Different contracts produce identical graphs, enabling evasion attacks (Finding 1).
- The representation vocabulary is limited to 47 hardcoded entries (including one duplicate), producing inconsistent, degraded, and fabricated output (Finding 2).
- The most critical structural element of the targeted attack, the external caller, is absent from the graph schema. The $C \rightarrow W$ edge is generated inconsistently across contracts implementing the same vulnerability, meaning the model receives incomplete representations with no signal that anything is missing (Finding 3).
- A fully exploitable contract is predicted as safe, directly caused by the absent $C \rightarrow W$ edge (Finding 4).

These are representation problems, not model problems. The neural network never receives the information needed to make the correct decision.

10.2 What This Means for Reported Accuracy

Our results establish the existence of representation-layer failures and provide one confirmed example in which such a failure leads directly to a misclassification. In particular, Finding 4 shows that a fully exploitable contract can be labelled safe when the graph extraction process omits information required to encode the reentrancy relationship.

At the same time, our study does not measure how frequently these failures occur in real-world contract populations, nor does it estimate their aggregate effect on detector accuracy. Quantifying that impact would require a large-scale corpus study in which contracts are re-extracted and re-evaluated under conditions designed to expose these failure modes.

The extent to which real-world contract populations fall outside the extractor’s 47-entry whitelist, and the resulting effect on detection performance, therefore remains an open empirical question. Existing evaluation protocols primarily measure model performance on supplied graph representations; they do not explicitly test the representation-layer failure modes documented in this paper.

10.3 Scope: Why GNNSCVulDetector

This study focuses on GNNSCVULDETECTOR [8] because it is the foundational system in this line of work, widely cited as the basis for subsequent GNN-based detectors, and provides full open-source access to both the graph extraction pipeline and the trained model, making all findings independently reproducible. The failures documented in this paper are experimentally verified only for GNNSCVULDETECTOR. Whether similar vulnerabilities exist in other graph extraction pipelines remains an open empirical question. We hypothesize that systems relying on comparable surface-level matching mechanisms may exhibit related failure modes, but evaluating this requires direct testing of those systems and is left for future work.

10.4 Generalizability as Future Work

We therefore do not claim that the vulnerabilities identified here are widespread. Rather, we present a reproducible methodology that can be applied to additional GNN-based detectors to determine whether similar representation-layer failures occur elsewhere. Applying the renaming-based evaluation protocol used here to other systems would establish whether the failure modes documented in this paper are specific to GNNSCVULDETECTOR’s implementation or reflect a broader pattern in graph extraction design.

10.5 Representation Before Model

Most research in smart contract security focuses on improving the model: better architectures, better training, more data. Our findings show that the more urgent problem is upstream, since if the representation is broken, the model cannot compensate. Future work should build representations that capture what code does rather than what code looks like: graphs built from Abstract Syntax Trees instead of name matching, control-flow analysis to detect the true order of external calls and state updates, data-flow analysis to trace values independent of naming, and explicit modelling of cross-contract calling relationships so the external caller is represented rather than omitted. Learned code embeddings such as CodeBERT are a plausible complement, replacing a fixed vocabulary with representations that generalise across naming conventions. Evaluation protocols must be extended accordingly, to cover out-of-whitelist naming, structural transformation, and cross-contract relationships.

These techniques share a common goal: representing program behaviour rather than lexical appearance. An AST-based representation captures syntactic structure independent of identifier naming, control-flow analysis captures execution ordering, and data-flow analysis tracks value dependencies across variables and functions. Together, these approaches reduce sensitivity to superficial code transformations while preserving the semantic relationships relevant to vulnerability detection.

11 Conclusion

This paper demonstrates that the graph representations used by GNN-based smart contract vulnerability detectors do not faithfully capture semantic differences between contracts. We show experimentally that structurally different contracts produce identical graphs, that graph construction is governed by a 47-entry whitelist that produces inconsistent, degraded, and fabricated output, that the $C \rightarrow W$ edge encoding the external caller is generated inconsistently across contracts meaning the graph schema fails to reliably represent the attack's active agent, and that a fully exploitable contract is predicted as safe as a direct consequence.

These failures are silent: the pipeline produces a graph, the model produces a label, and no existing evaluation protocol checks whether the graph faithfully represents the source contract. Existing evaluation protocols do not explicitly evaluate the failure modes documented in this paper.

Closing this gap requires changes at the representation layer: moving from surface-level pattern matching toward semantic analysis of what contracts do, and evaluation protocols that expose the failure modes documented here. Model-level improvements alone are insufficient.

References

- 1 P. K. Adjei et al. A graph attention network-based multi-agent reinforcement learning framework for robust detection of smart contract vulnerabilities. *Scientific Reports*, 15:29810, 2025.
- 2 H. Al-Harbi. Detecting anomalies in blockchain transactions using spatial-temporal graph neural networks. *Advances in Management and Intelligent Technology*, 1(1), 2025.
- 3 J. J. de Leon et al. Smart contract security in decentralized finance: Enhancing vulnerability detection with reinforcement learning. *Applied Sciences*, 15(11):5924, 2025.
- 4 M. Paramesha, N. L. Rane, and J. Rane. Artificial intelligence, machine learning, deep learning, and blockchain in financial and banking services. *PUMRJ*, 1(2), 2024.
- 5 S. Xu et al. DeFiGuard: A price manipulation detection service in DeFi using graph neural networks. *IEEE Transactions on Services Computing*, 17(6), 2024. doi:10.1109/TSC.2024.3489439.
- 6 Y. Zhang et al. Penetrating the hostile: Detecting DeFi protocol exploits through cross-contract analysis. *IEEE Xplore*, 2025.
- 7 Z. Zhang et al. Scalable blockchain fraud detection using spatial-temporal graph neural networks, 2024.
- 8 Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural networks. In *IJCAI*, pages 3283–3290, 2020. doi:10.24963/IJCAI.2020/454.