BugSweeper: Function-Level Detection of Smart Contract Vulnerabilities Using Graph Neural Networks

Uisang Lee, Changhoon Chung, Junmo Lee, Soo-Mook Moon*

Seoul National University
{uisang.lee, ch.chung, junmo.lee, smoon}@snu.ac.kr

Abstract

The rapid growth of Ethereum has made it more important to quickly and accurately detect smart contract vulnerabilities. While machine learning-based methods have shown some promise, many still rely on rule-based preprocessing designed by domain experts. Rule-based preprocessing methods often discard crucial context from the source code, potentially causing certain vulnerabilities to be overlooked and limiting adaptability to newly emerging threats. We introduce BugSweeper, an end-to-end deep learning framework that detects vulnerabilities directly from the source code without manual engineering. BugSweeper represents each Solidity function as a Function-Level Abstract Syntax Graph (FLAG), a novel graph that combines its Abstract Syntax Tree (AST) with enriched control-flow and data-flow semantics. Then, our two-stage Graph Neural Network (GNN) analyzes these graphs. The first-stage GNN filters noise from the syntax graphs, while the second-stage GNN conducts high-level reasoning to detect diverse vulnerabilities. Extensive experiments on real-world contracts show that BugSweeper significantly outperforms all state-of-the-art detection methods. By removing the need for handcrafted rules, our approach offers a robust, automated, and scalable solution for securing smart contracts without any dependence on security experts.

Extended version — https://arxiv.org/abs/2512.09385

Introduction

Blockchain is a decentralized and distributed ledger that enables open participation without third-party intermediaries and has attracted significant interest from academia and industry (Krichen et al. 2022). Ethereum extended blockchain capabilities by introducing smart contracts, which are digital agreements written in Solidity code that automatically execute transactions (Buterin et al. 2013). However, these smart-contract programs introduce potential security vulnerabilities that can be exploited by malicious attackers. According to an empirical study (Durieux et al. 2020) that analyzed 47,368 smart contracts, many vulnerabilities, such as reentrancy and unchecked low-level calls, were reported. In particular, the DAO attack (Daian 2016) exploited a reentrancy vulnerability to steal 3.6 million Ether (valued at $60 million at the time). Furthermore, smart contracts cannot be modified once deployed, unlike traditional software. Fixing a deployed contract typically requires deleting the original and redeploying an updated version, which can be both inconvenient and costly. For these reasons, it is crucial to thoroughly verify the security of smart contracts before deployment.

A variety of code-analysis techniques have been proposed to detect vulnerabilities, including static analysis (Tikhomirov et al. 2018; Feist, Greico, and Groce 2019; Wang et al. 2024), symbolic execution (Luu et al. 2016; Mueller 2017; Mossberg et al. 2020), and dynamic execution (Jiang, Liu, and Chan 2018; Choi et al. 2022; Liu et al. 2018). However, these conventional methods heavily rely on manually crafted expert rules, making them ineffective against the rapid emergence of new vulnerabilities that bypass predefined patterns.

To overcome these drawbacks, researchers have increasingly leveraged deep learning models for smart contract vulnerability detection. For instance, Peculiar (Wu et al. 2021) and ReVulDL (Zhang et al. 2023) utilize GraphCodeBERT (Guo et al. 2021), while TMP (Zhuang et al. 2021) and AME (Liu et al. 2021) apply Graph Neural Networks (GNNs). These deep learning-based approaches can reduce analysis time and minimize reliance on expert-crafted rules. However, in a given smart contract, only a small fraction of the code is typically involved in a vulnerability. This observation motivates extracting vulnerability-specific code fragments for training vulnerability detection models. Existing deep learning methods incorporate preprocessing steps for extraction. However, these still depend on rigid, rule-based heuristics, resulting in several limitations:

- **Restricted Scope**: They overlook vulnerabilities that are not captured by predefined rules. For example, novel variations of reentrancy attacks that deviate from established heuristics may remain undetected.
- **Poor Generalization**: Deep learning models relying on narrow, rule-based preprocessing cannot identify other vulnerability types (e.g., *unchecked low-level calls*, *arithmetic errors*) that do not fit existing patterns.
- **Information Loss**: If preprocessing rules are inaccurately specified, crucial details in the original code may be lost.

*Corresponding author.

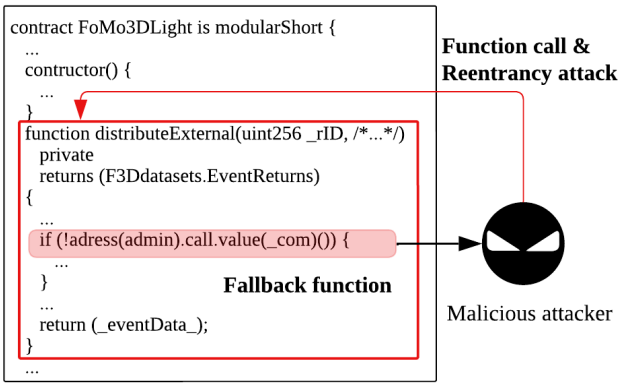

Figure 1: An example scenario demonstrating a reentrancy attack.

In this paper, we introduce BugSweeper, a novel graph-based framework for detecting vulnerabilities in smart contracts. BugSweeper employs a GNN trained on our proposed Function-Level Abstract Syntax Graph (FLAG) representation. It effectively identifies various vulnerability types in a fully automated and data-driven manner without relying on predefined expert rules, thereby overcoming the limitations of previous rule-based preprocessing methods.

Figure 1 provides an example of a reentrancy attack. In this scenario, the `distributeExternal` function uses `call.value()` to send Ether to an external address, which in turn triggers the fallback function of a malicious contract. The fallback function—a default function that executes when no other function matches, or when Ether is received—then repeatedly calls `distributeExternal`, causing it to re-enter and execute again before the previous invocation completes. Such cases illustrate that many security vulnerabilities in smart contracts arise from unsafe inter-function interactions. Inspired by this observation and motivated by the limitations of current rule-based preprocessing methods, we propose analyzing smart contracts at the function level for more precise vulnerability detection.

While analyzing code at the function level provides precise details, it neglects important connections between functions and still leads to redundant information. To solve this, we first convert Solidity code into abstract syntax trees (ASTs) and then divide these trees into separate function-level subtrees. We enrich these subtrees by adding edges representing function calls and variable references, creating FLAGs. Additionally, we introduce a parameter called *coverage* to control the number of inter-function connections included.

However, higher coverage settings introduce extra noise, complicating the learning process. To address this, we develop a two-stage GNN architecture. The first stage, a Code Graph Neural Network (CGNN), summarizes AST node details at the function level to mitigate noise and produces pooled function graphs. Then, the second-stage GNN analyzes these pooled graphs to capture meaningful relationships between functions.

Our extensive experiments demonstrate that BugSweeper significantly outperforms both traditional static analysis tools and existing deep learning models. Overall, it detects diverse vulnerability classes in an end-to-end, data-driven manner without relying on predefined expert rules.

This paper makes the following contributions:

- **A Novel Data-Driven, Multi-Class Vulnerability Detection Framework**: We introduce BugSweeper, the first unified and rule-free framework that detects multiple vulnerability classes directly from code structure. Experiments conducted on three distinct vulnerability types—(i) Reentrancy, (ii) Unchecked Low-Level Calls, and (iii) Time Manipulation—demonstrate the effectiveness of our function-level, data-driven approach that generalizes to diverse vulnerabilities.
- **Function-Level Representation with Two-Stage GNN Architecture**: We introduce FLAG, a specialized graph representation that captures the rich semantic and structural features of Solidity code relevant to security analysis. Furthermore, our two-stage GNN architecture—comprising a CGNN and a second-stage GNN—effectively reduces AST-level noise and captures meaningful relationships among pooled function graphs.

Related Work

Traditional Smart Contract Analysis Methods. Traditional methods, which rely on static analysis and symbolic execution, have been widely used to identify vulnerabilities in smart contracts. Specifically, static analysis-based methods detect vulnerability patterns by analyzing code, data flow, and control flow using predefined rules, without executing the program. Static analysis-based tools for smart contract vulnerability detection include SmartCheck (Tikhomirov et al. 2018) and Slither (Feist, Greico, and Groce 2019). Symbolic execution-based methods identify vulnerabilities by tracking program execution paths and collecting symbolic values for conditions along those paths. Representative tools include Oyente (Luu et al. 2016), Osiris (Torres, Schütte, and State 2018), Manticore (Mossberg et al. 2020), and Mythril (Mueller 2017). Recent methods, such as Slise (Wang et al. 2024), employ a hybrid approach that combines static analysis with symbolic execution.

Deep Learning-Based Smart Contract Analysis Methods. Various deep learning-based methods have been developed for vulnerability detection. TMP (Zhuang et al. 2021) converts contracts into symbolic graphs, refines them by removing less important nodes, and normalizes the graph to apply Graph Neural Networks (GNNs). Furthermore, AME (Liu et al. 2021) combines expert patterns with TMP. In contrast, Peculiar (Wu et al. 2021) and ReVulDL (Zhang et al. 2023) focus on a contract's data flow, targeting critical variables within a data-flow graph to analyze reentrancy vulnerabilities using the GraphCodeBERT model (Guo et al. 2021). However, current deep learning approaches have not fully escaped the rigidity of earlier methods. Many still depend on rule-based components, such as the explicit expert patterns in AME or the heuristic-based graph simplification in TMP. This reliance on heuristics and specialization fundamentally limits their applicability to various vulnerabilities.

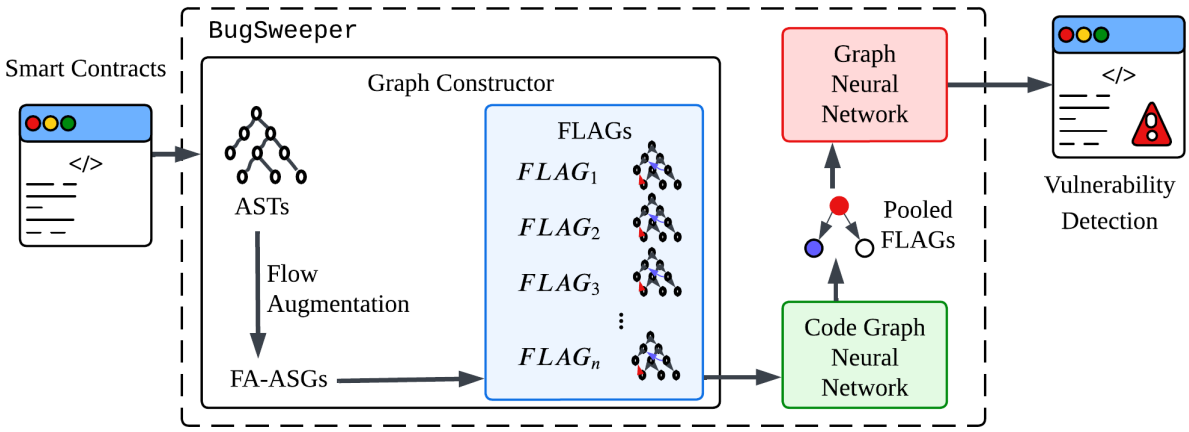

Figure 2: The overall architecture of BugSweeper. BugSweeper parses each contract into an AST, augments it with control-flow and data-flow edges to form Function-Level Abstract Syntax Graphs (FLAGs), and then applies a two-stage GNN: a Code Graph Neural Network, followed by a second-stage GNN.

BugSweeper

In this section, we introduce BugSweeper, a GNN-based framework for detecting smart contract vulnerabilities, and describe its architecture in three parts: the Graph Constructor, the Code Graph Neural Network, and the Second-Stage Graph Neural Network.

Graph Constructor

Previous research showed that AST-based approaches perform well on many code-related tasks (Zhang et al. 2019), and FA-AST (Wang et al. 2020) demonstrated that adding data-flow and control-flow details to an AST helps the model outperform a basic AST approach. Building on this, we use an abstract syntax graph that integrates data and control flow.

Figure 3 illustrates how we construct FLAGs. First, we use the Solidity compiler *solc* as a parser to convert the source code into an AST. To enable function-level extraction, we mark the nodes in subtrees whose root is a `FunctionDefinition` or `VariableDeclaration`. A basic AST fails to capture critical program execution details, so we augment it with supplementary edges to form a flow-augmented abstract syntax graph (FA-ASG). Specifically, we introduce control-flow and data-flow edges to create a richer semantic representation. Our FA-ASG consists of three distinct edge categories: the original structural edges from the AST, along with the newly added control-flow and data-flow edges, as detailed below.

- Basic edges capture the fundamental structure of the AST. Specifically, *Child* edges link parent nodes to their child nodes, while *Parent* edges connect child nodes back to parent nodes, thereby improving overall node connectivity (Allamanis, Brockschmidt, and Khademi 2018).
- Data-flow edges illustrate how data moves through a program. Specifically, *ReferencedDeclaration* edges indicate the use of previously defined variables or functions. *FunctionReturnParameter* edges directly link a `FunctionDefinition` node to the data it returns. *SuperFunction* edges represent function-overriding relationships. *Assignment* edges capture the process of assigning data to variables that have already been declared.
- Control-flow edges show how the execution sequence flows in `IfStatement`, `LoopStatement`, and `NextStatement` nodes. For an `IfStatement`, *CondTrue* indicates the path when the condition is true, while *CondFalse* shows the path when it is false. A `LoopStatement` includes *WhileExecution* and *ForExecution* edges that point to the condition, and *WhileNext* and *ForNext* edges leading to the statements executed once the condition is met. `NextStatement` edges represent the sequential ordering of statements.

Next, the contract is divided into multiple subgraphs, where each subgraph represents a single function. To model the dependencies between these functions, we introduce a hyperparameter termed coverage. This parameter controls the depth of neighborhood expansion when constructing each function's graph, determining the extent to which subgraphs of called functions or referenced variables are included. This concept is illustrated in Figure 4.

For instance, a coverage of 1 includes only the subgraph of the target function itself. A coverage of 2 expands the graph to include the subgraphs of all directly called functions and variables (the 1-hop neighborhood). At a coverage of 3, the graph further incorporates the subgraphs of functions and variables called by that second level (the 2-hop neighborhood). Through this iterative expansion, we generate enriched FLAGs that serve as our final units for analysis.

Code Graph Neural Network (CGNN)

In this section, we explain how CGNN converts each FLAG into a Pooled FLAG, reducing unnecessary graph noise through pooling techniques. Each node in a FLAG carries textual attributes (e.g., `FunctionDefinition`). To convert the textual node attributes into vector representations

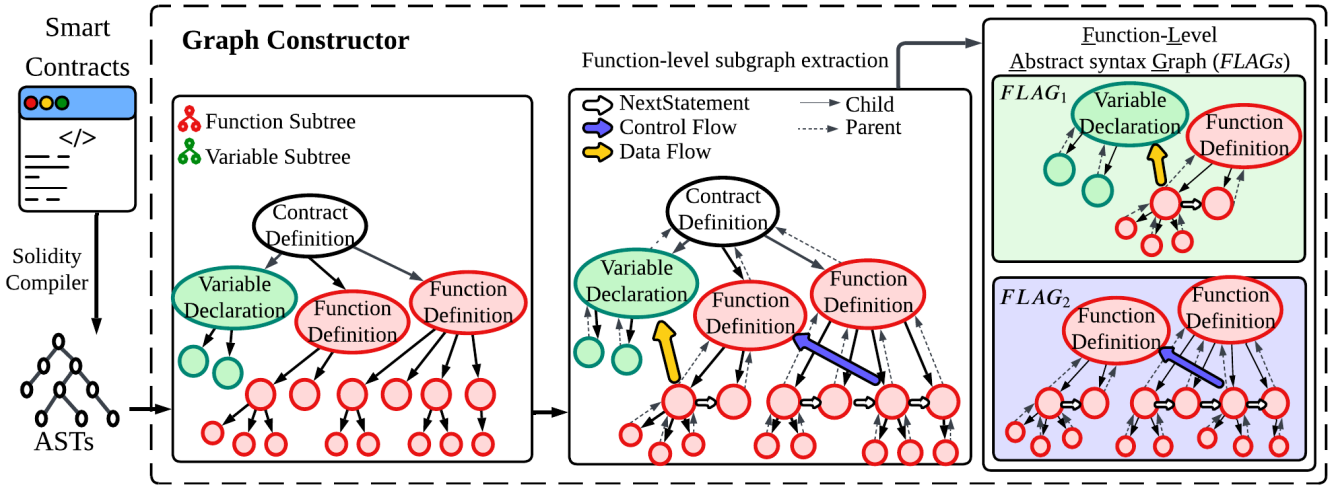

Figure 3: An overview of FLAG (Function-Level Abstract Syntax Graph) construction. The Graph Constructor enriches the AST with explicit control-flow (blue) and data-flow (yellow) edges. We then extract function-level subgraphs, called FLAGs, which represent target functions while preserving key connections to related code. The example uses coverage = 2, so each FLAG also includes directly connected functions and variables.

suitable for neural network input, we apply the BPE tokenizer (Sennrich, Haddow, and Birch 2016).

After tokenizing the node attributes, we apply GraphSAGE (Hamilton, Ying, and Leskovec 2017), an efficient, inductive GNN, to generate detailed node embeddings. GraphSAGE updates each node representation by aggregating features from its neighboring nodes using the following message-passing mechanism:

$$h_v^k \leftarrow AGG^k\left(\left\{\left(h_u^{(k-1)} : u \in \mathcal{N}(v)\right)\right\}\right) \oplus h_v^{k-1}, \quad (1)$$

$$h_v^k \leftarrow \sigma(\mathbf{W}^k \cdot h_v^k), \quad (2)$$

where k is the layer number for message-passing depth K, h_v^0 is the initial node feature encoded via a BPE tokenizer, and $\mathcal{N}(v)$ refers to the neighborhood function, which maps each node v to its set of neighboring nodes in the graph. AGG aggregates feature vectors of neighboring nodes, and $\oplus$ represents the concatenation operation that merges the aggregated message with h_v^{k-1}. Table 1 shows the overall architecture of CGNN.

For graph-level vulnerability classification, individual node embeddings must be aggregated into a single representation. While global mean and max pooling are commonly used in many GNN applications, they are suboptimal for our FLAGs. These naive methods fail to distinguish between core functional nodes and auxiliary nodes introduced by external function calls or variable references, often yielding noisy and uninformative graph embeddings.

Several advanced graph pooling methods have been proposed, such as TopKPool (Gao and Ji 2019), SAGPool (Lee, Lee, and Kang 2019), and ASAPool (Ranjan, Sanyal, and Talukdar 2020). TopKPool and SAGPool select and retain nodes by scoring them based on node features and attention mechanisms, respectively. On the other hand, ASAPool groups nodes into clusters based on local subgraph scores. Although effective, TopKPool and SAGPool often face scalability issues on large graphs and can unintentionally discard critical structural information. While ASAPool addresses some of these issues, its clustering process is computationally expensive, limiting its practicality for large-scale applications.

Layer	Input Size	Output Size	Activation Function
SAGEConv	512	1024	ReLU
SAGEConv	1024	1024	ReLU
SAGEConv	1024	1024	-

Table 1: CGNN architecture

Rather than relying on computationally expensive clustering or learned node-scoring mechanisms, we propose a deterministic and efficient semantic pooling method called Code Graph Pool (CGPool). It groups nodes based on their syntactic roles in the source code—for example, merging all nodes that belong to the same function declaration or variable definition into a single supernode.

This design preserves key high-level relationships by reconnecting these supernodes according to their original control-flow and data-flow links. The resulting structure, which we term the Pooled FLAG, is a compact yet faithful abstraction of the program's logic (as shown in Figure 4). CGPool effectively integrates the benefits of previous pooling strategies. Like ASAPool, it maintains important hierarchical structures but without the computational cost. Similar to TopKPool and SAGPool, CGPool reduces graph complexity without the risk of information loss from aggressive node selection.

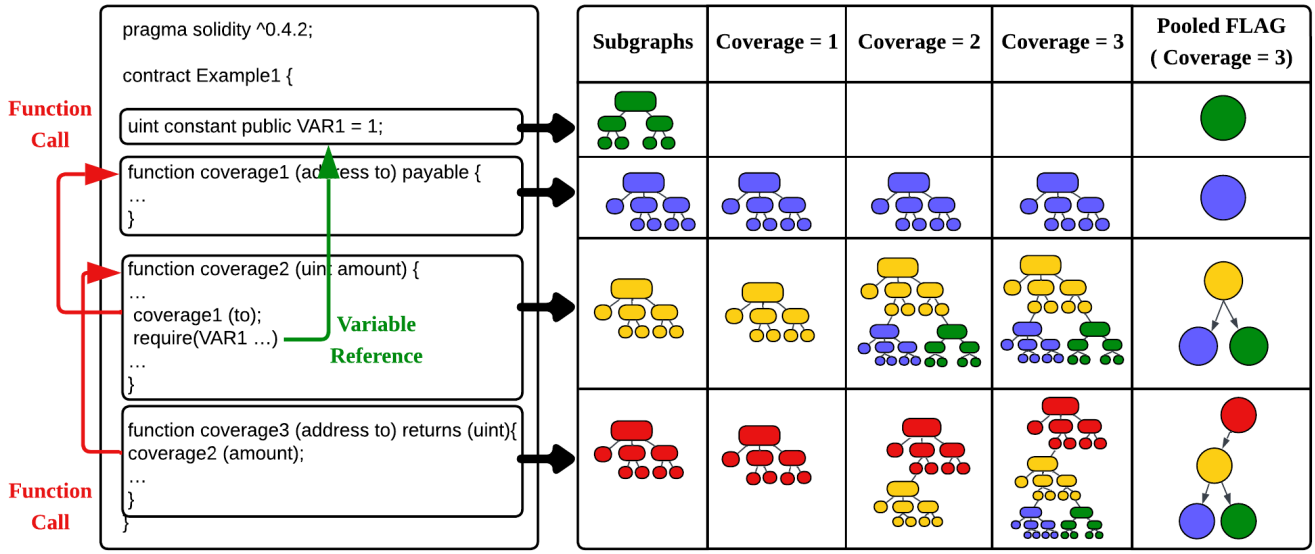

Figure 4: Solidity code is parsed into an AST and decomposed into function- and variable-level subgraphs. Higher coverage incorporates more inter-function connections. The graph for `coverage1` stays the same across coverage levels, as it does not involve any function calls or variable references. Colors in the figure are solely for distinguishing the subgraphs.

Layer	Input Size	Output Size	Activation Function	Heads
GATConv	1024	1024	ReLU	4
GATConv	1024	1024	ReLU	1
GATConv	1024	1024	ReLU	1
Linear1	1024	1024	ReLU	–
Linear2	1024	1024	ReLU	–
Linear3	1024	C	–	–

Table 2: Second-stage GNN and classifier architecture. C refers to the number of vulnerability classes.

Second-Stage Graph Neural Network

After the pooling step, we obtain a simplified FLAG where each node represents a meaningful group of elements from the original code, such as functions or variables. This compact structure significantly reduces the number of nodes while preserving essential structural information. Because of this reduction, we can efficiently apply a GNN to capture higher-level relationships among the supernodes.

We use a Graph Attention Network (GAT) (Veličković et al. 2018) to process the Pooled FLAG. GAT assigns dynamic weights to edges based on their importance, allowing the model to focus more on critical connections—such as those between function nodes or between a function and its related variables. This produces rich node embeddings that encode both local and contextual information. After the GAT layers, we apply a global readout function to summarize the entire graph into a single vector. This vector reflects the vulnerability patterns identified in the contract and serves as the input to the classifier.

The classifier consists of three linear layers: the first two use ReLU activation, and the final layer applies a softmax function to produce class probabilities. It is trained using numerous examples of source code paired with their corresponding labels to predict whether a given contract contains vulnerable functions. Although we demonstrate BugSweeper's effectiveness across three vulnerability types, our model can be extended to detect a broader range of vulnerabilities. Table 2 summarizes the architecture of the second-stage GNN and classifier used in BugSweeper.

Experiments

Datasets and Experimental Setup

Dataset. We use the AME dataset (Liu et al. 2021) to compare the performance of various methods for detecting reentrancy vulnerabilities. Other methods were either limited to detecting only reentrancy vulnerabilities or did not specify their preprocessing steps, making it challenging to reproduce results for additional vulnerability types. The AME dataset contains 1,224 smart contracts, which we divide into 719 contracts for training and 505 contracts for testing, following the original experimental setup for accurate performance comparisons.

Additionally, we use the SmartBugs Wild dataset (Durieux et al. 2020), consisting of 47,398 Solidity files with a total of 203,716 contracts, to evaluate our model's performance on multi-class vulnerability detection tasks and conduct ablation studies. The SmartBugs Wild dataset labels contracts as vulnerable if three or more heuristic methods agree on the presence of a vulnerability, significantly reducing false positives and ensuring reliable ground truth labels. The dataset is structured based on this consensus approach as follows:

- **Reentrancy**: Reentrancy happens when a contract function calls an external contract before updating its state, which can allow attackers to re-enter the function mul-

tiple times. A contract is labeled with this vulnerability only if confirmed by four or more tools, ensuring high confidence for this critical bug.

- **Unchecked Low-Level Calls**: This category includes vulnerabilities related to functions like `call()` or `send()` that fail without proper checks. For this type, a vulnerability is labeled as positive if detected by a consensus of three or more tools.
- **Time Manipulation**: In Solidity, `block.timestamp` can be manipulated by miners, posing security risks. These vulnerabilities are labeled as positive if detected by a consensus of at least three tools.

Experimental Setup. We evaluate performance using macro-averaged Precision, Recall, and F1-score (Wu et al. 2021). Macro-averaging calculates metrics for each class individually and then averages them, making it suitable for vulnerability detection tasks, which typically involve highly imbalanced datasets. This ensures equal contribution of each class to the overall evaluation. We train all models using the Adam optimizer (Kingma and Ba 2014) with a learning rate of 1e-4 and weight decay of 1e-5. We train for up to 500 epochs with a batch size of 64. For robustness, each experiment was repeated four times with different random seeds, and all table entries report the mean over these runs. In most comparisons, the performance improvements are statistically significant (paired t-test, $p \leq 0.05$). The mean $\pm$ standard deviation results and full paired t-test statistics are given in the Appendix.

Reentrancy Detection Results

We evaluate our proposed method, BugSweeper, against four baseline vulnerability detection tools—Slither (Feist, Greico, and Groce 2019), SmartCheck (Tikhomirov et al. 2018), Mythril (Mueller 2017), and Slise (Wang et al. 2024)—as well as four deep learning-based techniques, TMP (Zhuang et al. 2021), AME (Liu et al. 2021), Peculiar (Wu et al. 2021), and ReVulDL (Zhang et al. 2023).

Table 3 presents the results of our experiments, highlighting several important findings. First, traditional methods (Slither, SmartCheck, Mythril, Slise) exhibited poor performance in detecting reentrancy vulnerabilities, as evidenced by their low recall scores. Second, deep learning-based methods achieved significantly better performance overall. Notably, our function-level approach, BugSweeper, outperformed all other models on the AME dataset. BugSweeper demonstrated an improvement in precision (up to 99.87%) and achieved the highest overall F1-score (98.57%), outperforming the closest competitor by approximately 3.1%. These results indicate that BugSweeper significantly reduces both false positives and false negatives compared to existing approaches.

Overall, our findings confirm the effectiveness of BugSweeper's function-level, rule-independent approach for detecting reentrancy vulnerabilities. By constructing FLAGs that integrate both syntax structure and control- and data-flow information, BugSweeper adapts to diverse vulnerability patterns—resulting in greater flexibility, robustness, and accuracy in smart contract security analysis.

Approach	Reentrancy		
	Precision (%)	Recall (%)	F1 (%)
Slither	94.74	34.62	50.70
SmartCheck	88.57	55.36	68.13
Mythril	84.48	47.12	60.49
Slise	95.56	45.26	61.43
TMP	87.06	83.74	85.36
AME	95.45	95.38	95.42
Peculiar	92.58	94.40	93.48
ReVulDL	92.95	94.62	93.74
BugSweeper (ours)	**99.87**	**97.35**	**98.57**

Table 3: Performance comparison of various approaches for detecting reentrancy vulnerabilities. The first four methods represent traditional rule-based detection tools, while the remaining approaches are deep learning-based.

Multi-class Vulnerability Detection Results

To evaluate BugSweeper's capability in detecting multiple vulnerability types, we conducted multi-class classification experiments focusing on three categories: (i) Reentrancy, (ii) Unchecked Low-Level Calls, and (iii) Time Manipulation. Additionally, we investigated the impact of different GNN configurations on detection performance.

Table 4 illustrates that our proposed two-stage BugSweeper models outperform the single-stage baseline models across all vulnerability types. Specifically, the two-stage approach significantly improves performance, effectively capturing complex vulnerability patterns.

Among the tested configurations, the combination of GraphSAGE in the first stage and GAT in the second stage (SAGE + GAT) achieved the highest performance, obtaining an F1-score of 91.61% for reentrancy, 80.15% for unchecked low-level calls, and 79.63% for time manipulation. These results demonstrate how GraphSAGE and GAT play distinct but synergistic roles. GraphSAGE excels at aggregating information across large, detail-rich graphs, while GAT's attention mechanism sharpens focus on the most critical features in the pooled, higher-level representation.

However, we observed comparatively lower detection performance for unchecked low-level calls and time manipulation vulnerabilities relative to reentrancy. This difference is due to data imbalance, as noted in (Durieux et al. 2020). The smaller number of examples for these vulnerabilities makes it more challenging for the model to learn effective representations, resulting in somewhat lower performance across our evaluation metrics.

Ablation Studies

Component Study. To validate the effectiveness of BugSweeper's architecture, we conducted a detailed ablation study, with results shown in Table 4. The goal of this experiment was to understand how each component contributes to overall performance.

First, we created a baseline model representing a simpler, single-stage GNN. This baseline does not include our proposed CGPool or the two-stage GNN structure. Instead, it

Architecture	Model	Reentrancy			Unchecked Low-Level Calls			Time Manipulation		
		Precision (%)	Recall (%)	F1 (%)	Precision (%)	Recall (%)	F1 (%)	Precision (%)	Recall (%)	F1 (%)
BugSweeper (single-stage GNN)	GAT	90.28	78.23	84.77	81.86	68.33	71.43	90.07	63.88	74.72
	SAGE	87.77	78.97	83.11	83.32	54.97	66.20	88.69	64.68	74.77
BugSweeper	GAT + SAGE	82.24	82.82	82.46	83.20	66.67	74.02	78.83	61.89	69.33
	GAT + GAT	86.42	85.38	85.66	89.01	60.81	72.16	81.14	63.88	72.50
	SAGE + SAGE	87.59	90.25	88.89	83.41	69.01	75.38	84.58	67.46	75.06
	SAGE + GAT	90.91	**92.31**	**91.61**	85.70	**75.44**	**80.15**	89.27	69.44	**79.63**

Table 4: Performance analysis of different GNN configurations within the BugSweeper framework on the SmartBugs Wild dataset; the analysis was conducted with a coverage of 4. The results confirm that our two-stage architecture significantly outperforms the single-stage baseline, with the SAGE + GAT model achieving the highest F1-scores. Bolded results are the best and statistically significant ($p \leq 0.05$, paired t-test).

Model	Pool	Multi-class Vulnerability		
		Precision (%)	Recall (%)	F1 (%)
BugSweeper (CGNN only)	TopKPool	87.74	72.48	78.71
	SAGPool	83.91	69.29	75.13
	ASAPool	89.85	74.02	80.32
BugSweeper (SAGE + GAT)	TopKPool	84.77	68.86	75.29
	SAGPool	88.59	69.55	77.10
	ASAPool	87.37	78.60	82.41
	CGPool (ours)	91.27	**84.21**	**87.32**

Table 5: Multi-class vulnerability detection performance for different pooling methods on BugSweeper variants. BugSweeper (CGNN only) denotes a single-stage CGNN with GraphSAGE. Bolded results are the best and statistically significant.

uses a standard GNN architecture (GAT or GraphSAGE) followed by a global pooling layer to produce the final graph representation.

Next, we evaluated our complete two-stage BugSweeper architecture, testing several combinations of GNN models across the two stages. For instance, a configuration labeled GAT + SAGE indicates that we use GAT for the first stage (CGNN) and GraphSAGE in the second stage to analyze the pooled graph representation.

Our experiments provide two key insights. First, the two-stage architecture consistently outperforms the single-stage baseline, confirming that the second-stage GNN effectively refines and enhances the representations produced by the first stage. Second, the combination of GraphSAGE in the first stage, followed by GAT in the second stage, consistently achieves the highest overall F1-score. This result demonstrates a complementary synergy: GraphSAGE initially captures a broad structural context, and GAT's attention mechanism precisely identifies critical features within the reduced, second-stage graph.

Pooling Method. To evaluate the effectiveness of our domain-specific CGPool, we replace CGPool in BugSweeper with three widely used graph pooling methods—TopKPool (Gao and Ji 2019), SAGPool (Lee, Lee, and Kang 2019), and ASAPool (Ranjan, Sanyal, and Talukdar 2020)—and evaluate their performance on multi-class vulnerability detection (Table 5). TopKPool scores nodes using learned projection vectors and retains only the highest-scoring subset, while SAGPool extends this idea by incorporating neighbor information via graph convolutions. ASAPool, in contrast, hierarchically clusters local subgraphs to capture rich structural patterns. For comparison, we evaluate a simplified variant that removes the second-stage GNN, leaving a single CGNN. In this single-stage setting, we apply one of these pooling methods and then use global mean pooling on its output to obtain the final graph representation.

The results in Table 5 demonstrate two key findings. First, substituting CGPool with any of the pooling methods in the BugSweeper architecture yields lower F1-scores (75.13–82.41%) than our CGPool baseline (87.32%), confirming that CGPool's AST-aware semantic clustering is better suited to code analysis.

Second, our experiments validate the effectiveness of the two-stage GNN architecture for this task. Pooling methods demonstrate improved performance with a second GNN layer, except for TopKPool, indicating that this second stage can refine the features from the initial pooling step to enhance detection performance.

In summary, CGPool not only achieves the highest F1-score but also maintains balanced precision (91.27%) and recall (84.21%) compared to other methods, showing its domain effectiveness for function-level vulnerability detection.

Conclusion

In this paper, we present BugSweeper, a function-level framework for detecting vulnerabilities in smart contracts. BugSweeper has two core components: a Graph Constructor that builds Function-Level Abstract Syntax Graphs (FLAGs) from the source code, and a two-stage Graph Neural Network (GNN) that first mitigates noise and then performs high-level reasoning over the extracted graphs. In addition, our domain-specific pooling method, Code Graph Pool (CGPool), effectively reduces information loss during graph abstraction. Experimental results demonstrate that BugSweeper not only outperforms existing approaches in detecting reentrancy vulnerabilities but also remains effective across multiple vulnerability categories. Ultimately, our framework enhances the accuracy and robustness of vulnerability detection, contributing to more secure and reliable smart-contract ecosystems.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by Korean Government [Ministry of Science and ICT (MSIT)] under Grant RS-2023-00208245, 30%; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by Korean Government (MSIT) under Grant 2021-0-00180, 40%; in part by the Information Technology Research Center (ITRC) support Program Supervised by IITP under Grant IITP-2021-0-01835, 10%; and in part by IITP under the Graduate School of Artificial Intelligence Semiconductor Grant IITP-2025-RS-2023-00256081, 10%; and in part by Hyundai Motor Chung Mong-Koo Foundation, 10%.

References

Allamanis, M.; Brockschmidt, M.; and Khademi, M. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

Buterin, V.; et al. 2013. Ethereum white paper. *GitHub repository*, 1: 22–23.

Choi, J.; Kim, D.; Kim, S.; Grieco, G.; Groce, A.; and Cha, S. K. 2022. SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, 227–239. IEEE Press. ISBN 9781665403375.

Daian, P. 2016. Analysis of the DAO exploit.

Durieux, T.; Ferreira, J. a. F.; Abreu, R.; and Cruz, P. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, 530–541. New York, NY, USA: Association for Computing Machinery. ISBN 9781450371216.

Feist, J.; Greico, G.; and Groce, A. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, 8–15. IEEE Press.

Gao, H.; and Ji, S. 2019. Graph U-Nets. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 2083–2092. PMLR.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 1025–1035. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781510860964.

Jiang, B.; Liu, Y.; and Chan, W. K. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, 259–269. New York, NY, USA: Association for Computing Machinery. ISBN 9781450359375.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krichen, M.; Ammi, M.; Mihoub, A.; and Almutiq, M. 2022. Blockchain for Modern Applications: A Survey. *Sensors*, 22(14).

Lee, J.; Lee, I.; and Kang, J. 2019. Self-Attention Graph Pooling. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 3734–3743. PMLR.

Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; and Roscoe, B. 2018. ReGuard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, 65–68. New York, NY, USA: Association for Computing Machinery. ISBN 9781450356633.

Liu, Z.; Qian, P.; Wang, X.; Zhu, L.; He, Q.; and Ji, S. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, 2751–2759.

Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; and Hobor, A. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 254–269. New York, NY, USA: Association for Computing Machinery. ISBN 9781450341394.

Mossberg, M.; Manzano, F.; Hennenfent, E.; Groce, A.; Grieco, G.; Feist, J.; Brunson, T.; and Dinaburg, A. 2020. Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, 1186–1189. IEEE Press. ISBN 9781728125084.

Mueller, B. 2017. Mythril-Reversing and bug hunting framework for the Ethereum blockchain.

Ranjan, E.; Sanyal, S.; and Talukdar, P. 2020. Asap: Adaptive structure aware pooling for learning hierarchical graph representations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, 5470–5477.

Sennrich, R.; Haddow, B.; and Birch, A. 2016. Neural Machine Translation of Rare Words with Subword Units. In Erk, K.; and Smith, N. A., eds., *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 1715–1725. Berlin, Germany: Association for Computational Linguistics.

Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; and 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*, WETSEB '18, 9–16. New York, NY, USA: Association for Computing Machinery. ISBN 9781450357265.

Torres, C. F.; Schütte, J.; and State, R. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, 664–676. New York, NY, USA: Association for Computing Machinery. ISBN 9781450365697.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

Wang, W.; Li, G.; Ma, B.; Xia, X.; and Jin, Z. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree . In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 261–271. Los Alamitos, CA, USA: IEEE Computer Society.

Wang, Z.; Chen, J.; Wang, Y.; Zhang, Y.; Zhang, W.; and Zheng, Z. 2024. Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts. *Proc. ACM Softw. Eng.*, 1(FSE).

Wu, H.; Zhang, Z.; Wang, S.; Lei, Y.; Lin, B.; Qin, Y.; Zhang, H.; and Mao, X. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 378–389. IEEE.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; and Liu, X. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 783–794. IEEE.

Zhang, Z.; Lei, Y.; Yan, M.; Yu, Y.; Chen, J.; Wang, S.; and Mao, X. 2023. Reentrancy Vulnerability Detection and Localization: A Deep Learning Based Two-phase Approach. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22. New York, NY, USA: Association for Computing Machinery. ISBN 9781450394758.

Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; and He, Q. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20. ISBN 9780999241165.