Smart Contract
Vunerability Detector

Puneet Parameswaran [*1] **Soham Purushan** [*1]

Abstract

Smart contracts are pivotal in blockchain applications, yet their security vulnerabilities have led to significant financial losses. Traditional manual audits and rule-based systems are resource-intensive and error-prone, motivating the use of advanced machine learning techniques for vulnerability detection. Initially, we investigated a broad approach using Graph Neural Networks (GNNs) to classify smart contracts as secure or vulnerable, representing code structures through abstract syntax tree (AST)-based graphs and applying Graph Convolutional Networks (GCN), Graph Isomorphism Networks (GIN), and Temporal Message Passing Networks (TMP). GIN models demonstrated superior accuracy in binary classification, showcasing effectiveness in capturing structural patterns of vulnerabilities. To refine our analysis further, we concentrated specifically on reentrancy vulnerabilities—one of the most notorious attack vectors in smart contracts, exemplified by the DAO exploit. We utilized a Temporal Message Passing Network (TMPNetwork), explicitly modeling both control and data flow as graph edges with temporal and semantic attributes. The TMPNetwork iteratively updated node representations using temporal edge sequences and employed a gated readout mechanism for accurate classification. Our targeted model achieved a remarkable accuracy of 95% and an F1-score of 0.93, significantly outperforming baseline methods. This integrated approach demonstrates a powerful advancement in automated smart contract security analysis.

[*]Equal contribution [1]University of North Carolina at Chapel Hill, USA. Correspondence to: Puneet Parameswaran <puneet@email.edu>, Soham Purushan <sopur@ad.unc.edu>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

1 Introduction

The advancement of blockchain technology has elevated smart contracts as critical components to run a streamline environment for financial security and development. These self-executing digital contracts ensure transparency and efficiency; however, they introduce significant security vulnerabilities. Due to the immutable nature of blockchain technology, vulnerabilities exploited in deployed smart contracts can lead to irreversible financial losses, an example would be incidents DAO hack exploiting reentrancy vulnerabilities. Thus, forming a project to improve security of smart contracts prior to deployment.

Traditional approaches to smart contract security auditing rely on manual code review, which is resource-intensive, expensive, and often inadequate due to the complexity and volume of modern smart contracts. Automated detection methods have become essential to address these problems, offering scalable, efficient, and reliable identification of potential vulnerabilities.

In our project, we first explore a broad-based automated vulnerability detection approach focusing on Graph Neural Networks (GNNs), particularly Graph Isomorphism Networks (GIN), to identify whether smart contracts contain vulnerabilities. GNNs naturally align with program code analysis due to their ability to capture structural and semantic properties represented as graph-based data structures, such as Control Flow Graphs (CFGs) and Abstract Syntax Trees (ASTs). To further enhance detection accuracy and depth the scope of our work, we narrowed our focus specifically to reentrancy vulnerabilities, employing a specialized Temporal Message Passing Network (TMPNetwork). TMPNetwork incorporates critical temporal and structural features, effectively capturing the nuanced dependencies that characterize reentrancy vulnerabilities.

Through extensive experimentation, our TMPNetwork-based approach demonstrated significant performance improvements over traditional rule-based methods, achieving high detection accuracy and robustness on a single type of vulnerability. By integrating generalized GNN-based detection with specialized temporal modeling, our research highlights the potential of graph-based deep learning techniques in automating and enhancing smart contract security analyses, allowing us to build future models for other types of vulnerabilities

Core Components and Technologies Our project integrates a robust set of computational frameworks and machine learning tools to effectively automate vulnerability detection in smart contracts. We primarily employed Python as our development environment, with the use of libraries such as `PyTorch` for neural network modeling, `NumPy` for numerical computation, `Scikit-learn` for performance evaluation, and `Matplotlib` for insightful visualization of results.

Smart contracts were represented in structured graph formats to capture complex syntactic and semantic relationships within the source code. Initially, Abstract Syntax Trees (ASTs) extracted from Solidity smart contracts were parsed and transformed into graph data structures using `NetworkX` and `PyTorch Geometric (PyG)`. Each node in these graphs represents a critical code element (such as functions, variables, and key opcodes), enriched with numerical features reflecting code semantics. Edges model both control flow and data flow dependencies, effectively encoding the execution logic and structural interactions of the contracts.

In our initial phase, we explored multiple Graph Neural Network (GNN) architectures:

- **Graph Convolutional Network (GCN):** Initially employed due to its efficiency in aggregating information across neighboring nodes.
- **Graph Isomorphism Network (GIN):** Selected for its expressive power in capturing subtle graph differences, ultimately providing superior detection performance.
- **Graph Attention Network (GAT):** Evaluated for its ability to weigh relationships dynamically, although it showed less consistent results compared to GIN.

Through experimentation, we identified the GIN model as the most effective general-purpose architecture, achieving robust detection across multiple types of vulnerabilities.

Recognizing the complexity and feature variability associated with basic vulnerability classification, we refined our focus specifically to the *reentrancy* vulnerability—a high-impact security flaw historically exploited in high-profile incidents such as the DAO attack. This strategic decision streamlined node feature representation, reduced complexity, and significantly enhanced model accuracy by narrowing the learning scope.

For this focused analysis, we implemented a specialized **Temporal Message Passing Network (TMPNetwork)**. The TMPNetwork advances traditional GNN approaches by explicitly modeling temporal and sequential dependencies among code elements, which are critical for accurately detecting vulnerabilities like reentrancy, where execution order matters.

Key components of our TMPNetwork implementation include:

- **Edge-type Embedding:** Capturing distinct semantic meanings of interactions between nodes.
- **Temporal Message Propagation:** Sequentially updating node representations following the temporal execution traces inherent in smart contract behavior.
- **Gated Readout Mechanism:** Combining initial and final node states to yield robust vulnerability predictions.

Model training utilized the Adam optimizer alongside the Binary Cross-Entropy (BCE) loss function, developing effective convergence. We retrieved successful metrics such as accuracy, precision, recall, and F1-score, supplemented by visual analytics including confusion matrices, ROC curves, and precision-recall plots to provide nuanced insights into model behavior.

Overall, this framework demonstrates the usefulness of using graph neural networks for automated smart contract security auditing, offering a scalable and precise alternative to manual vulnerability assessments.

2 Methodology

This project employs Graph Neural Networks (GNNs) to detect specific smart contract vulnerabilities (Reentrancy, Timestamp Dependency, etc.). GNNs are a class of deep learning models designed to operate directly on graph-structured data, enabling them to learn from both node features and the relational structure (edges) of the graph. This makes them suitable for analyzing smart contracts, where program elements and their dependencies can be naturally represented as graphs. Specifically, this work focuses on implementing and evaluating the Temporal Message Propagation (TMP) network proposed by Zhuang *et al.* for vulnerability detection. The methodology encompasses vulnerability-specific graph construction and normalization, feature engineering, the TMP model architecture, and the training/prediction pipeline.

2.1 Graph Generation and Normalization

The analysis process begins with parsing smart contracts into **Abstract Syntax Trees (ASTs)**, capturing essential syntactic and semantic details required for vulnerability assessment. These ASTs are then systematically transformed into directed graphs where nodes correspond to key functional elements within contracts, and edges represent critical relationships such as control flow, data flow, and sequential dependencies.

Initial Graph Extraction. Nodes identified during parsing are classified into:

- **Major Nodes:** Critical elements including function invocations and built-in functions vital to vulnerability detection.
- **Secondary Nodes:** Representing supporting operations and variables necessary for detailed flow analysis.
- **Fallback Nodes:** Specifically representing fallback mechanisms when relevant.

Edges capture sequential execution order and semantic interactions (e.g., Forward (FW), Read/Write (RG), Access Control (AC)).

Graph Normalization. To emphasize central functionalities and standardize diverse graph representations:

- Secondary and fallback nodes are eliminated, aggregating their attributes into neighboring major nodes.
- Edges initially connected to removed nodes are redirected towards corresponding major nodes.

Feature Vector Construction. Each remaining node has its initial feature vector constructed by concatenating one-hot encodings reflecting characteristics such as node type and access control, forming an initial comprehensive feature matrix.

Temporal Edge List. Normalized edges are explicitly ordered to capture the temporal execution sequence, a critical aspect for detecting vulnerabilities sensitive to operational order.

2.2 Graph Neural Networks Employed

We explored three distinct Graph Neural Network architectures for vulnerability detection:

- **Graph Convolutional Network (GCN):** This foundational architecture aggregates neighborhood information to effectively capture local graph structures, serving as our baseline model.
- **Graph Isomorphism Network (GIN):** GIN provides enhanced discriminative capabilities by capturing nuanced graph structures and node attributes, leading to improved accuracy in distinguishing vulnerable from secure contracts.
- **Temporal Message Propagation Network (TMP):** TMP specifically integrates temporal sequencing information into node updates, crucial for vulnerabilities heavily dependent on execution order.

Our comparative analysis revealed the superior capability of the TMP model, demonstrating higher precision and accuracy in identifying vulnerabilities across various types of smart contract issues.

2.3 Training and Evaluation Pipeline

We adopted a rigorous training protocol for all GNN models, employing **Binary Cross-Entropy (BCE)** loss optimized using the **Adam optimizer**. Evaluation metrics included: Accuracy, Precision, Recall, F1-score.

2.4 Vulnerability-Specific Graph Generation and Normalization

The foundation of the approach is the transformation of smart contract source code (SC) into a normalized, directed graph $G = (V, E)$ tailored to highlight patterns relevant to a specific vulnerability type.

Initial Graph Extraction. The source code is parsed to identify critical program elements based on the target vulnerability. These are mapped to an initial node set V', including:

- **Major Nodes** (M, e.g., `.call.value`)
- **Secondary Nodes** (S, e.g., relevant variable operations)
- **Fallback Node** (F, if applicable)

Edges E' are constructed to represent control flow, data flow, and sequential dependencies, each defined as a tuple

$$(V'_s, V'_e, o_k, t_k),$$

denoting start node, end node, temporal order o_k, and semantic type t_k (e.g., FW, RG, AC).

Graph Normalization. To emphasize key components and standardize structure (cf. Sec. 3.2 of Zhuang *et al.*):

1. Eliminate Secondary and Fallback nodes, aggregating their features into the nearest Major node(s).
2. Redirect all edges incident on eliminated nodes to the corresponding Major node.

The result is a final node set V, where each $V_i \in V$ corresponds to an original Major node M_i with aggregated features.

Feature Vector Construction. For each $V_i \in V$, form an initial hidden state vector $\mathbf{h}_i^{(0)}$ by concatenating one-hot encodings of:

(node type, access control, in-features, out-features, ...).

Collect all $\mathbf{h}_i^{(0)}$ into the matrix $\mathbf{H}^{(0)}$.

Temporal Edge List. Denote the normalized edges as an ordered list

$$E = \{ e_1, \dots, e_N \},$$

where $e_k = (s_k, e_k, t_k)$ carries the integer indices of start node s_k, end node e_k, and semantic type t_k. The list order $k = 1, \dots, N$ encodes temporal sequence.

2.5 Temporal Message Propagation (TMP) Network

Given the graph $G = (\mathbf{H}^{(0)}, E)$, the TMP network predicts a vulnerability score $\hat{y}$ in two phases.

Message Propagation Phase. Iteratively update node states along edges in temporal order $k = 1 \dots N$:

$$\begin{aligned}
\mathbf{h}_{s_k} &= \mathbf{H}^{(k-1)}[s_k], \\
\mathbf{h}_{e_k}^{\text{prev}} &= \mathbf{H}^{(k-1)}[e_k], \\
\\
\mathbf{t}_k^{\text{emb}} &= \text{Embed}(t_k), \\
\mathbf{x}_k &= \mathbf{h}_{s_k} \,\|\, \mathbf{t}_k^{\text{emb}}, \\
\mathbf{m}_k &= W_k\, \mathbf{x}_k + \mathbf{b}_k, \\
\widehat{\mathbf{h}}_{e_k} &= \tanh\bigl(U\,\mathbf{m}_k + Z\,\mathbf{h}_{e_k}^{\text{prev}} + \mathbf{b}_1\bigr), \\
\mathbf{H}^{(k)} &= \mathbf{H}^{(k-1)} \text{ with row } e_k \text{ replaced by } \widehat{\mathbf{h}}_{e_k}.
\end{aligned}$$

Readout Phase. After N steps, let $\mathbf{H}^{(N)} = \mathbf{H}^T$. For each node i:

$$\mathbf{s}_i = \mathbf{h}_i^T \,\|\, \mathbf{h}_i^0.$$

Compute gating vectors via two-layer MLPs:

$$\begin{aligned}
\mathbf{g}_i &= \text{softmax}\bigl(W_g^{(2)} \tanh(W_g^{(1)}\mathbf{s}_i + \mathbf{b}_g^{(1)}) + \mathbf{b}_g^{(2)}\bigr), \\
\mathbf{o}_i &= \text{softmax}\bigl(W_o^{(2)} \tanh(W_o^{(1)}\mathbf{s}_i + \mathbf{b}_o^{(1)}) + \mathbf{b}_o^{(2)}\bigr).
\end{aligned}$$

Aggregate to form the final prediction:

$$\hat{y} \;=\; \sum_{i=1}^{|V|} \sigma\bigl(\mathbf{o}_i \odot \mathbf{g}_i\bigr),$$

where $\odot$ is element-wise multiplication and σ is the sigmoid.

2.6 Training and Prediction Pipeline

Distinct TMP models are trained per vulnerability type.

Training. Given a training set $\{(G_j, y_j)\}$, we minimize the Binary Cross-Entropy loss

$$\mathcal{L} = -\bigl[y_j \log \hat{y}_j + (1 - y_j) \log(1 - \hat{y}_j)\bigr]$$

by updating all model parameters θ using the Adam optimizer with learning rate $\eta = 0.001$. Concretely, each parameter step is

$$\theta \;\leftarrow\; \theta \;-\; \eta\, \nabla_\theta \mathcal{L}\,.$$

Prediction. For a new contract:

1. Generate G_{new} using the graph pipeline.
2. Load trained parameters θ^*.
3. Compute $\hat{y}_{\text{new}} = \text{TMP}(G_{\text{new}};\, \theta^*)$.

3 Results

We trained the TMPNetwork for reentrancy detection over 100 epochs (Adam, $\eta = 0.001$). Below are the key results on our validation set.

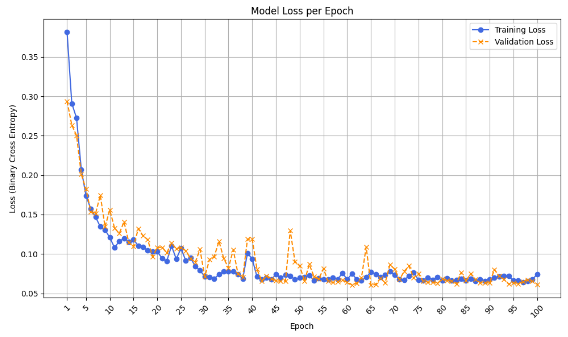

Figure 1: Training and validation loss (binary cross-entropy) across 100 epochs. Notice that the curves stabilize around epoch 30 with a final loss of ≈ 0.07.

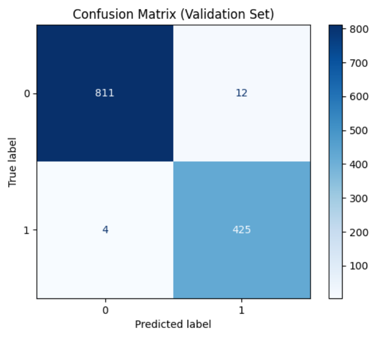

Figure 2: Confusion matrix for reentrancy detection. True negatives: 811, false positives: 12; false negatives: 4, true positives: 425. Very few misclassifications demonstrate strong calibration.

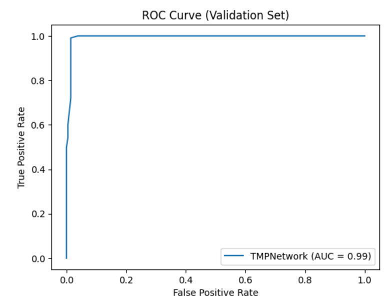

Figure 3: ROC curve (AUC = 0.9942). The steep rise at the start and the curve hugging the top-left corner affirm that TMPNetwork is an extremely strong classifier on the validation data.

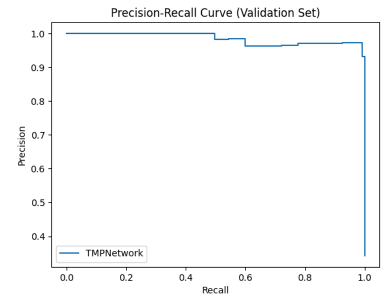

Figure 4: Precision–Recall curve. Precision remains very high over almost the entire recall range, indicating very few false positives and robust positive-class detection.

4 Future Work

We propose extending the TMP framework into a *unified multi-label, multi-task GNN* capable of detecting several vulnerabilities—reentrancy, timestamp dependency, integer overflow, delegatecall, and others—within a single model. The current datasets from **(author?)** (1) were curated for *single-error* training; combining them requires both representation and architecture changes.

4.1 Graph Representation

1. **Reuse an existing extractor**. Apply one of the present scripts (e.g. the timestamp extractor) to every contract, acknowledging that some features relevant to other errors may be absent.

2. **Design a more encompassing extractor**. Introduce additional node and edge types so the graph encodes all patterns needed for every vulnerability class. This demands extra engineering and an investigation into common error signatures that generalizes across tasks.

4.2 Model Modifications

- **Multi-Label Head** Replace the final scalar output with `nn.Linear(hidden_dim, N)` followed by element-wise sigmoid, where N is the number of vulnerability types. Train using multi-hot targets and `BCEWithLogitsLoss`.

- **Multi-Task Heads** Keep the shared TMP backbone but attach N independent two-layer MLP heads (each ending in a sigmoid). Compute a separate BCE loss for each head and sum (or average) the losses during back-propagation.

4.3 Lightweight Ensemble Pipeline

An alternative that avoids re-engineering graphs is to orchestrate the existing specialised TMP models:

1. **Pre-processing** – run each `AutoExtractGraph` + `graph2vec` pipeline (reentrancy, timestamp, overflow, delegatecall . . .).

2. **Inference** – load the corresponding pretrained TMP model for each graph and obtain its probability score.

3. **Aggregation** – collect the scores (e.g. {`reentrancy`: 0.85, `timestamp`: 0.12, . . . }) into a single report or decision module.

We hopefully want to explore both the unified architecture and the ensemble pipeline to determine which offers the best balance between development effort, runtime efficiency, and detection performance.

References

[1] Y. Zhuang *et al.*, "Smart Contract Vulnerability Detection Using Graph Neural Networks," IJCAI-20.

[2] K. Xu, W. Hu, J. Leskovec, S. Jegelka, "How Powerful are Graph Neural Networks?," ICLR 2019.

[3] "Multilayer perceptron," https://en.wikipedia.org/wiki/Multilayer_perceptron.

[4] J. Brownlee, "A Gentle Introduction to Generative Adversarial Networks," Machine Learning Mastery, Jun. 2018, https://machinelearningmastery.com/gentle-introduction-to-generative-adversarial-ne

[5] "Graph Convolutional Networks: Introduction to GNNs," Medium (Oct. 2018), https://medium.com/towards-data-science/graph-convolutional-networks-introduction-to-gnn

[6] "How to Design the Most Powerful Graph Neural Network," Medium (Feb. 2020), https://medium.com/towards-data-science/how-to-design-the-most-powerful-graph-neural-network-3d18b07a6e66