# 

**Final Project Phase 3: Machine Learning – AI-Driven Smart Contract Auditing Using ML**

Swapnali Dashrath (sdashrath@oakland.edu)

Oakland University

CSI6160 - Machine Learning

Prof. Dr Mohammad Wardat.

December 8, 2024.

1. **Introduction:**

#### Dataset Expansion:

To address the data imbalance and improve the accuracy of the smart contract auditing model, we adopted a systematic approach to programmatically generate Solidity files with both **secure** and **insecure** contracts. Here's a detailed description of the methodology:

#### Modified Approach

### 1. Understanding the Dataset Requirements

* **Secure Contracts**: These contracts adhere to best practices, include robust access controls, avoid known vulnerabilities, and use established libraries like OpenZeppelin.
* **Insecure Contracts**: These contracts intentionally include vulnerabilities such as reentrancy, improper access controls, integer overflows, and uninitialized state variables.
* **Balanced Dataset**: The dataset was balanced with an equal number of secure and insecure contracts to prevent model bias toward the majority class.

#### Proposed Solution

### 2. Programmatic Generation Strategy

To generate diverse and representative Solidity contracts, we implemented the following steps:

#### (a) Defining Vulnerability Patterns for Insecure Contracts

* Reentrancy Vulnerability**:**
  + External calls before state changes.
  + Contracts mimicking real-world exploits (e.g., DAO attack).
* Unrestricted Access**:**
  + Missing onlyOwner modifiers for sensitive functions.
* Unchecked Arithmetic**:**
  + Lack of safe arithmetic operations (e.g., missing SafeMath library usage).
* Uninitialized State Variables**:**
  + Leaving state variables uninitialized, leading to default values being exploited.
* Fallback Misuse**:**
  + Contracts relying on fallback() without restrictions.

#### (b) Defining Best Practices for Secure Contracts

* Proper Access Control**:**
  + Using onlyOwner or similar modifiers to restrict sensitive functions.
* Reentrancy Mitigation:
  + Using Checks-Effects-Interactions pattern and ReentrancyGuard.
* Safe Arithmetic:
  + Employing SafeMath for arithmetic operations.
* Input Validation:
  + Adding checks for valid ranges, non-zero addresses, etc.

#### (c) Programmatic Code Templates

#### Created parameterized Solidity templates with placeholders for introducing variations in:

#### Function and variable names.

#### Contract structure (e.g., number of functions, modifiers).

#### Logic complexity (e.g., nested control flows).

#### Example Template for an Insecure Contract:

python

Copy code

insecure\_template = """

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract {CONTRACT\_NAME} {{

mapping(address => uint256) public balances;

function deposit() public payable {{

balances[msg.sender] += msg.value;

}}

function withdraw() public {{

uint256 amount = balances[msg.sender];

require(amount > 0, "Insufficient balance");

// Reentrancy vulnerability: external call before state update

(bool success, ) = msg.sender.call{{value: amount}}("");

require(success, "Transfer failed");

balances[msg.sender] = 0;

}}

}}

"""

Example Template for a Secure Contract:

python

Copy code

secure\_template = """

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract {CONTRACT\_NAME} is ReentrancyGuard {{

mapping(address => uint256) public balances;

function deposit() public payable {{

balances[msg.sender] += msg.value;

}}

function withdraw() public nonReentrant {{

uint256 amount = balances[msg.sender];

require(amount > 0, "Insufficient balance");

balances[msg.sender] = 0; // Update state before external call

(bool success, ) = msg.sender.call{{value: amount}}("");

require(success, "Transfer failed");

}}

}}

"""

#### (d) Automating Code Generation

#### Python Script: Used Python to programmatically generate Solidity files by:

#### Substituting placeholders in templates with randomized values.

#### Writing files to a specified directory (./contracts).

#### Ensuring sufficient diversity by modifying:

#### Variable/function names.

#### Vulnerability placements.

#### Contract structures.

#### 3. Post-Generation Validation

#### Syntax Validation: Used solc (Solidity compiler) to ensure all generated files were syntactically correct.

#### Diversity Check: Verified that generated contracts contained enough variation in structure and vulnerability patterns.

#### Manual Review: Randomly sampled a subset of files for manual inspection.

#### 4. Impact on Model Performance

#### Dataset Expansion: Increased the dataset size to 50 secure and 50 insecure contracts. (Total 316 solidity file considering variations)

#### Balanced Representation: Addressed the issue of over-representation by ensuring equal class distribution.

#### Improved Accuracy: The expanded dataset allowed the model to achieve an accuracy of 85.94%, compared to prior performance.

#### 5. Suggestions for Further Improvements

#### Expand Vulnerability Patterns: Include additional vulnerabilities like timestamp dependencies, unauthorized delegate calls, or DoS attacks.

#### Real-World Data: Incorporate real-world contracts from GitHub or Etherscan, labeled using automated vulnerability scanners.

#### Advanced Models: Experiment with deep learning approaches for better semantic understanding.

### 8. DATA AVAILABILITY

1. The datasets used in this project are available at **https://github.com/sdashrath/SmartContractAuditingV1.git.**
2. The repository includes the labeled dataset, scripts for preprocessing, Automation Script for generating solidity files and model training.
3. This dataset is shared under the **Creative Commons Attribution 4.0 International License**, allowing reuse with proper attribution.

**References**

1. **Wood, G.** (2014). *Ethereum: A Secure Decentralized Generalized Transaction Ledger*. Retrieved from [**https://ethereum.org/en/whitepaper/**](https://ethereum.org/en/whitepaper/)
   * This whitepaper explains the Ethereum blockchain, smart contracts, and consensus mechanisms such as Proof of Stake (PoS).
2. **DeepAI.org.** (2021). *Anomaly Detection with AI*. Retrieved from [**https://deepai.org/machine-learning-glossary-and-terms/anomaly-detection**](https://deepai.org/machine-learning-glossary-and-terms/anomaly-detection)
   * A detailed explanation of AI-based anomaly detection and its application in various industries, including blockchain and decentralized finance
3. Schar, F. (2021). **Decentralized Finance: On Blockchain- and Smart Contract-based Financial Markets**. *Federal Reserve Bank of St. Louis Review*, 103(2), 153-174.
   * This paper provides an in-depth overview of DeFi applications and their relevance in modern finance.
4. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778. <https://doi.org/10.1109/CVPR.2016.90>
5. Foundation for Learning Equality. *TF-IDF Vectorizer Documentation*. scikit-learn. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.TfidfVectorizer.html.
6. OpenZeppelin. *Solidity Smart Contract Auditing Standards*. Available at: <https://openzeppelin.com/>.

7. Mythril. *Mythril Tool for Smart Contract Security Analysis*. Available at: <https://github.com/ConsenSys/mythril>.

8. Brownlee, J. *How to Evaluate Machine Learning Algorithms*. Machine Learning Mastery, 2019. Available at: https://machinelearningmastery.com/how-to-evaluate-machine-learning-algorithms/.