DETECTING MALICIOUS SMART CONTRACT INTERACTION USING MACHINE LEARNING

A DESIGN PROJECT REPORT

Submitted by

ABINAYA J

ATCHAYA DURGA K

JANE BEULA A

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

K. RAMAKRISHNAN COLLEGE OF TECHNOLOGY

(An Autonomous Institution, affiliated to Anna University Chennai and Approved by AICTE, New Delhi)

SAMAYAPURAM – 621 112

NOVEMBER,2024

DETECTING MALICIOUS SMART CONTRACT INTERACTION USING MACHINE LEARNING

A DESIGN PROJECT REPORT

Submitted by

ABINAYA J(811722104002)

ATCHAYA DURGA K(811722104018)

JANE BEULA A(811722104060)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

K.RAMAKRISHNAN COLLEGE OF TECHNOLOGY

(An Autonomous Institution, affiliated to Anna University Chennai and Approved by AICTE, New Delhi)

SAMAYAPURAM – 621 112

NOVEMBER,2024.

K.RAMAKRISHNAN COLLEGE OF TECHNOLOGY
(AUTONOMOUS)
SAMAYAPURAM – 621 112

BONAFIDE CERTIFICATE

Certified that this project report titled **" DETECTING MALICIOUS SMART CONTRACT INTERACTION USING MACHINE LEARNING "** is the bonafide work of **ABINAYA J (811722104002), ATCHAYA DURGA K(811722104018), JANE BEULA A(811722104060)**who carried out the project under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr.A.DELPHIN CAROLINA RANI, M.E, Ph.D**.,**

HEAD OF THE DEPARTMENT

Department of CSE
K.Ramakrishnan College of Technology
(Autonomous)
Samayapuram – 621 112

SIGNATURE

Mr R VIGNESH KUMAR M.E.,

SUPERVISOR

ASSISTANT PROFESSOR
Department of CSE
K.Ramakrishnan College of Technology
(Autonomous)
Samayapuram – 621 112

Submitted for the viva-voce examination held on ……

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DECLARATION

We jointly declare that the project report on **"DETECTING MALICIOUS SMART CONTRACT INTERACTION USING MACHINE LEARNING "** is the result of original work done by us and best of our knowledge, similar work has not been submitted to **"ANNA UNIVERSITY CHENNAI"** for the requirement of Degree of **BACHELOR OF ENGINEERING**. This project report is submitted on the partial fulfilment of the requirement of the award of Degree of **BACHELOR OF ENGINEERING**.

Signature

_____

ABINAYA J

_____

ATCHAYA DURGA K

_____

JANE BEULA A

Place: Samayapuram

Date:

ACKNOWLEDGEMENT

It is with great pride that we express our gratitude and in-debt to our institution "**K.Ramakrishnan College of Technology (Autonomous)**", for providing us with the opportunity to do this project.

We are glad to credit honourable chairman **Dr. K.RAMAKRISHNAN**, **B.E.,** for having provided for the facilities during the course of our study in college.

We would like to express our sincere thanks to our beloved Executive Director **Dr. S. KUPPUSAMY, MBA, Ph.D.,** for forwarding to our project and offering adequate duration in completing our project.

We would like to thank **Dr. N. VASUDEVAN, M.E., Ph.D.,** Principal, who gave opportunity to frame the project the full satisfaction.

We whole heartily thanks to **Dr. A. DELPHIN CAROLINA RANI**, **M.E., Ph.D.,** Head of the department, **COMPUTER SCIENCE AND ENGINEERING** for providing his encourage pursuing this project.

I express my deep and sincere gratitude to my project guide **Mr R VIGNESH KUMAR**, **M.E.**, Department of **COMPUTER SCIENCE AND ENGINEERING,** for his incalculable suggestions, creativity, assistance and patience which motivated me to carry out this project.

I render my sincere thanks to Course Coordinator and other staff members for providing valuable information during the course.

I wish to express my special thanks to the officials and Lab Technicians of our departments who rendered their help during the period of the work progress.

ABSTRACT

Smart contracts have emerged as a key innovation, enabling decentralized and self-executing agreements without the need for intermediaries. However, the increasing complexity of smart contracts and their vulnerabilities have also led to a rise in security concerns, with malicious contracts posing a substantial threat to blockchain ecosystems. Detecting these malicious contracts is a critical challenge that requires innovative approaches.This project addresses the need for an automated system capable of classifying smart contracts as either benign or malicious based on their features. The primary objective of the project is to develop a machine learning-based classification system that can identify and flag potentially harmful smart contracts. The project begins by collecting a dataset of smart contract features, including function counts, modifier counts, self-destruct functions, and other relevant code attributes. These features are extracted from both benign and malicious contracts.A supervised machine learning approach is employed to train a classification model, with Random Forest being chosen as the algorithm due to its ability to handle complex datasets and produce accurate results.

TABLE OF CONTENTS.

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 OVERVIEW

The **Smart Contract Classification System** is a sophisticated machine learning-driven application designed to classify smart contracts as either benign or malicious. In recent years, the popularity of blockchain technology has surged, with smart contracts becoming an integral part of decentralized applications (dApps). However, with the growth in the use of smart contracts, the risk of malicious contracts being deployed has also increased. These contracts, if not carefully reviewed and audited, could have severe consequences, such as draining funds, altering or manipulating data, or introducing vulnerabilities that can be exploited by attackers.

This project aims to develop a machine learning-based solution to identify potential malicious behavior in smart contracts, thus enhancing the security and trustworthiness of blockchain applications. The system can automatically evaluate Solidity-based smart contracts (the most widely used programming language for writing smart contracts) based on key extracted features. It then classifies the contract into one of two categories: **benign** or **malicious**.

The core idea is to train a supervised machine learning model on a labeled dataset of known benign and malicious smart contracts, with each contract represented by a set of extracted features such as function count, modifier count, and the presence of dangerous operations like selfdestruct or require. Once trained, the system can predict the class of a new, unseen contract.

1.1.1 Key Features:

- **Feature Extraction:** The system extracts a variety of features from the smart contract code (Solidity files). These features represent characteristics such as the length of the contract, the number of functions, and other potentially dangerous elements like selfdestruction functions.
- **Prediction with Machine Learning:** The system uses a trained model to classify contracts into benign or malicious categories based on the extracted features. The model uses various machine learning algorithms to make predictions.
- **Web Interface:** The system provides an intuitive user interface built using Flask, which allows users to upload smart contract files and receive instant predictions about their nature (benign or malicious).

The significance of this project lies in its potential to safeguard blockchain applications, preventing costly exploits and ensuring that smart contracts are safe for use in decentralized systems. The system leverages cutting-edge machine learning techniques to offer an automated solution that enhances the efficiency of security audits for smart contracts.

1.2 TECHNOLOGY

1.2.1 Machine Learning

Machine learning is at the heart of the contract classification system. The core of the Smart Contract Classification System is machine learning. The classification task is approached as a supervised learning problem, where the system is trained on a dataset consisting of smart contracts labeled as benign or malicious. The machine learning algorithm learns patterns from the features of the smart contracts and generalizes those patterns to classify new, unseen contracts.

- **Feature Selection and Extraction:** The system uses a variety of features extracted from Solidity contracts. These features include:
    - **Contract Length:** The total number of lines of code in the smart contract. o **Function Count:** The number of functions defined in the contract.
    - **Modifier Count:** The number of modifiers (special functions that alter other functions' behavior).
    - o **Presence of Specific Operations:** Such as the use of selfdestruct or require, which may indicate risky behavior.

The machine learning model is trained using the following techniques:

- **Random Forest Classifier:** This ensemble method is used because it is robust, interpretable, and can handle large datasets effectively. It works by creating multiple decision trees during training and outputting the majority vote for classification.
- **Logistic Regression and Support Vector Machines :** These algorithms are also explored to compare their performance in distinguishing between benign and malicious contracts.

The model is then validated using **cross-validation**, ensuring that the system generalizes well to unseen data.

1.3 SYSTEM ARCHITECTURE

The architecture of this project can be broken down into several key layers and components, which interact with each other to provide a smooth user experience and accurate classification results.

1.3.1 Data Collection and Preprocessing Layer

The data collection layer gathers smart contract data from various blockchain platforms. These contracts are written in Solidity and are generally available from open-source repositories or blockchain explorers.

Once the contracts are collected, they are preprocessed:

- **Feature Extraction:** Key features, such as the number of functions, modifiers, and the presence of specific keywords (e.g., selfdestruct, require), are extracted. These features help the machine learning model to identify patterns indicative of malicious behavior.
- **Data Transformation:** The raw data (contract code) is transformed into numerical values, suitable for input into the machine learning model.

1.3.2 Machine Learning Model Layer

The core of the system, the machine learning layer, handles the training, evaluation, and prediction of smart contract classifications. This layer consists of:

- **Model Training:** The model is trained using labeled data of smart contracts, with both benign and malicious examples.
- **Hyperparameter Tuning:** Hyperparameters of the Random Forest model are tuned using cross-validation to achieve the best possible performance.
- **Model Evaluation:** The model is evaluated based on its accuracy, precision, recall, and F1-score, ensuring that it correctly classifies smart contracts into their respective categories.

Once trained, the model is saved in the form of a **Pickle** file (final_model.pkl), which can later be loaded into the Flask backend for making predictions.

1.3.3 Web Interface Layer (Frontend)

The frontend provides a user-friendly interface for interacting with the system. It is built using HTML, CSS, and JavaScript, ensuring that the platform is accessible and visually appealing. The frontend enables users to:

- **Upload Smart Contracts:** The users can upload .sol files (Solidity smart contracts) through a simple file upload form.
- **Receive Results:** After the smart contract is classified, the results (benign or malicious) are displayed to the user on the webpage.

The frontend interacts with the Flask backend via HTTP requests. When a user uploads a contract file, the frontend sends a request to the backend, which processes the file, makes a prediction using the machine learning model, and sends back the result.

1.3.4. Integration Layer

The integration layer connects the backend (where the model resides) with the frontend (user interface). This layer ensures that the system works seamlessly:

- **File Handling:** The Flask app accepts contract files from the frontend, validates them, and passes them to the model.
- **Prediction Process:** The model processes the extracted features and classifies the contract as benign or malicious.
- **Display Results:** The classification result is then sent back to the user interface, where it is presented to the user in a readable format.

1.4 SYSTEM FLOW DIAGRAM (HIGH-LEVEL OVERVIEW)

1. **User Interaction:**
    - Users visit the web interface and upload a smart contract file.
    - The contract file is sent to the backend Flask server for processing.

2. **Backend Processing:**
    - The Flask app extracts features from the contract file.
    - These features are then fed into the pre-trained machine learning model.
    - The model classifies the contract as benign or malicious.
3. **Result Display:**
    - The result is returned to the frontend, and the classification is displayed to the user.

The above architecture ensures that the system is scalable, modular, and efficient, providing a seamless experience for users while maintaining high accuracy in classifying smart contracts.

The **Smart Contract Classification System** integrates a variety of advanced techniques from **machine learning**, **natural language processing** , **data preprocessing**, and **blockchain security concepts**. These techniques help ensure that the system can accurately classify smart contracts into benign and malicious categories, improving the security of blockchain ecosystems. This section will provide a detailed discussion on the key methodologies and implementations that are used to achieve this goal.

1.4.1 Machine Learning Techniques

Machine learning is at the heart of the **Smart Contract Classification System**, allowing the model to learn from historical contract data to identify patterns indicative of malicious behavior. Several machine learning techniques were employed to ensure high accuracy and scalability.

a) Random Forest Classifier

- **Overview:** Random Forest is an ensemble learning method that builds multiple decision trees using different subsets of data and features. By combining the results from all individual trees, it reduces overfitting and provides more robust predictions compared to a single decision tree.
- **Implementation Details:** In the context of this project, Random Forest was utilized to classify smart contracts based on several key features such as contract length, function

count, and the presence of critical operations (e.g., selfdestruct, require). The algorithm creates multiple decision trees during the training phase, and each tree votes on whether the contract is benign or malicious. The final classification result is determined by the majority vote across all trees.

- **Benefits:** Random Forest is highly effective in situations where the dataset is large, and it performs well with noisy data. Additionally, it is resistant to overfitting and provides feature importance scores, which were instrumental in understanding which features contributed most to the classification of malicious contracts.

b) Logistic Regression

- **Overview:** Logistic Regression is a statistical method used for binary classification. It models the probability of the default class (benign or malicious) by fitting a logistic curve to the dataset. It calculates the probability that a given contract belongs to one class and uses a threshold (e.g., 0.5) to determine the class label.
- **Implementation Details:** Logistic Regression was employed as a benchmark model to compare the performance of more complex algorithms. The logistic model takes the features (such as modifier_count or fallback_count) and applies weights to them. These weights are optimized during training to minimize the error in classification.
- **Benefits:** Logistic Regression is simple and interpretable. It also offers easy calibration and can be a great starting point when working with new data. Its effectiveness in the project was validated through baseline results and comparisons with more complex models.

c) Support Vector Machines

- **Overview:** Support Vector Machines are supervised learning algorithms that construct a hyperplane to separate different classes in a high-dimensional feature space. SVM can handle both linear and non-linear data by using kernel tricks to map the data into a higher-dimensional space.
- **Implementation Details:** SVMs were tested to capture non-linear relationships between the features of the contracts. In the case of non-linearly separable data, the kernel trick was used to project the data into higher-dimensional spaces, allowing SVM to find the optimal hyperplane. The model was trained with a variety of kernels such as the radial

basis function and polynomial kernels to assess which kernel provided the best classification results.

- **Benefits:** SVM is known for its ability to handle high-dimensional data effectively, making it well-suited for this project. It works exceptionally well for complex data sets and performs well in terms of generalization.

d) Feature Engineering

- **Overview:** Feature engineering involves creating new input features or modifying existing features to improve the performance of machine learning models. For this project, feature engineering played a critical role in enhancing the predictive power of the model.
- **Implementation Details:** Features like function_count, modifier_count, selfdestruct_count, and require_count were initially derived from Solidity smart contract code. In addition, features like contract_length, and the number of external calls (e.g., call, delegatecall, etc.) were also engineered to capture more nuanced behaviors that might indicate malicious activity.
- **Benefits:** By carefully selecting and transforming the raw data into meaningful features, the system's performance significantly improved. The feature selection process was guided by domain knowledge about smart contracts and the typical characteristics of malicious code patterns.

1.4.2 Natural Language Processing Techniques

Natural Language Processing techniques are essential in this project to process the textual content of smart contract code. Although smart contract code is not the same as natural language text, it follows a set of syntactical and semantic rules that can be processed using NLP tools.

a) Tokenization

- **Overview:** Tokenization is the process of breaking down the smart contract code into smaller units (tokens), such as keywords, function names, variable names, operators, and control structures.

- **Implementation Details:** For each smart contract, the Solidity code was tokenized to separate the contract's components. For instance, function names like transfer() or selfdestruct() were identified as tokens, and special keywords like address or uint256 were treated as separate entities. This allows the model to understand the structural elements of the code.
- **Benefits:** Tokenization allowed the system to map each contract's raw code into a structured form that could be analyzed for patterns. It enabled the identification of important code snippets that could indicate malicious behavior.

b) Named Entity Recognition

- **Overview:** Named Entity Recognition is an NLP technique that identifies entities in the text, such as names of people, places, or specific operations. For this project, NER was adapted to identify specific operations and variables in Solidity code.
- **Implementation Details:** Using custom NER models, the system was able to identify smart contract-related entities such as function names (e.g., deposit(), withdraw(), selfdestruct()), contract addresses, and special terms like require, which are commonly associated with malicious contracts.
- **Benefits:** NER helped the model isolate important entities in the contract code, making it easier to identify features that are indicative of security vulnerabilities.

1.4.3 Data Preprocessing Techniques

Data preprocessing is an essential step in any machine learning pipeline. For this project, various preprocessing techniques were applied to ensure the data was clean, well-structured, and ready for analysis.

a) Data Normalization

- **Overview:** Data normalization is the process of scaling the features so that they fall within a specific range, typically between 0 and 1. It ensures that all features contribute equally to the model and prevents features with larger ranges from dominating the learning process.
- **Implementation Details:** Normalization was applied to numeric features like function_count, modifier_count, and contract length. These features had varying ranges,

so normalization ensured that no single feature would overshadow the others during model training.

- **Benefits:** Normalizing the features improved the performance of models such as Logistic Regression and Support Vector Machines, which are sensitive to the scale of input data.

b) Handling Missing Data

- **Overview:** Incomplete data, or missing values, can lead to inaccurate predictions or errors in model training. Proper handling of missing data is a critical step in ensuring model reliability.
- **Implementation Details:** Missing values were identified in the dataset and handled by imputation methods, such as replacing missing values with the mean or median of the feature, or using more advanced interpolation methods.
- **Benefits:** By addressing missing data, the system was able to maintain the integrity of the training process, resulting in a more accurate and reliable model.

1.4.4. Blockchain Security Techniques

Blockchain security is a core aspect of this project. The **Smart Contract Classification System** helps identify potentially dangerous or vulnerable contracts that could be exploited.

a) Smart Contract Security Audits

- **Overview:** Security audits are essential to ensuring the integrity of blockchain applications. By reviewing the smart contract code, auditors can identify vulnerabilities such as reentrancy attacks, integer overflows, or improper access controls.
- **Implementation Details:** The system uses classification to perform an initial screening of smart contracts, helping developers and auditors quickly identify contracts that may need a deeper audit. The features derived from the contracts, such as function calls and modifiers, are indicative of potential security risks.
- **Benefits:** This classification system provides an automated and efficient way to flag potentially malicious contracts, reducing the time required for security audits and minimizing the risk of exploits.

b) Blockchain Data Integrity

- **Overview:** Blockchain's decentralized nature ensures that once data is written to the ledger, it is immutable and tamper-resistant. However, vulnerabilities in smart contracts can undermine this integrity.
- **Implementation Details:** The system contributes to preserving blockchain integrity by ensuring that only safe, validated smart contracts are deployed, preventing malicious contracts from being executed on the blockchain.
- **Benefits:** By identifying and classifying smart contracts based on their security profile, this project helps maintain the integrity of the blockchain ecosystem, ensuring that only secure contracts are deployed.

By using these techniques, the Smart Contract Classification System ensures that contracts on blockchain networks are categorized accurately and efficiently, which is essential for enhancing security and trust in blockchain applications. This approach helps mitigate the risks associated with malicious or faulty smart contracts and contributes to safer decentralized applications.

CHAPTER 2

LITERATURE SURVEY

2.1 SURVEY ON MACHINE LEARNING

Title: Enhancing Smart-Contract Security through Machine Learning: A Survey of Approaches and Techniques

Authors: Keyang Gu, Junyi Wu, Yuanlong Cao

Year: 2023

Abstract: The article *"Enhancing Smart-Contract Security through Machine Learning: A Survey of Approaches and Techniques"* explores how machine learning (ML) can improve the security of smart contracts, which are vulnerable to various attacks like reentrancy and overflow issues. As smart contracts are crucial in blockchain ecosystems, especially Ethereum, their security is essential to prevent significant financial losses. The paper highlights how ML can be applied in both static and dynamic analysis to identify vulnerabilities, automate the detection of malicious behaviors, and improve the scalability of security measures. ML techniques such as supervised, unsupervised, and reinforcement learning are discussed, along with their applications in vulnerability detection, anomaly detection, and contract classification. However, challenges remain, such as limited labeled data, generalization across platforms, and the interpretability of models. The article concludes by suggesting that future research should focus on hybrid models, better data integration, and enhancing the transparency of ML-based security solutions for smart contracts.

2.2 COMPREHENSIVE SURVEY

2. **Title:** Vulnerability Detection in Smart Contracts: A Comprehensive Survey

Authors: Unspecified (Arxiv)

Year: 2024

Abstract: The article *"Vulnerability Detection in Smart Contracts: A Comprehensive Survey"* provides an in-depth overview of the various methods and tools developed to detect vulnerabilities in smart contracts, which are crucial components of

blockchain-based systems like Ethereum. Given their immutable and self-executing nature, smart contracts are highly susceptible to coding errors and security flaws that can lead to financial losses. The survey categorizes the approaches for vulnerability detection into static analysis, dynamic analysis, and hybrid methods. Static analysis examines the code without execution, looking for potential issues through techniques like symbolic execution, pattern matching, and formal verification. Dynamic analysis, on the other hand, observes the contract's behavior during execution, using methods like fuzzing and transaction tracing to identify vulnerabilities that only emerge in real-world interactions. Hybrid approaches combine both methods for more comprehensive coverage. The article also explores various automated tools and frameworks designed to identify common vulnerabilities such as reentrancy attacks, gas limit errors, and integer overflows. Despite the progress made, the survey highlights challenges, including the complexity of smart contract code, the lack of standardized test cases, and the difficulty of scaling detection tools. The paper concludes by emphasizing the need for more advanced techniques, better integration of tools, and the adoption of machine learning to improve vulnerability detection and enhance the overall security of smart contracts.

2.3 MALICIOUS CONTRACT DETECTION

3. **Title:** Malicious Contract Detection for Blockchain Network Using Lightweight Deep Learning

   Authors: Various

   Year: 2023

   Abstract: The article *"Deep Learning-Based Malicious Smart Contract and Intrusion Detection"* explores the application of deep learning techniques to enhance the security of blockchain networks by detecting malicious smart contracts and intrusions. As blockchain platforms like Ethereum increasingly rely on smart contracts for decentralized applications, they become vulnerable to various attacks, such as reentrancy, integer overflow, and logic flaws, which can be exploited by malicious actors. The paper proposes a deep learning-based approach to automatically identify these vulnerabilities by analyzing smart contract code and detecting abnormal patterns or potential threats. It highlights the use of deep neural networks (DNNs), recurrent neural networks (RNNs), and other advanced models to classify contracts as either benign or malicious based on their code behavior and transaction history. Additionally, the paper integrates intrusion

detection systems (IDS) to monitor real-time blockchain activities and identify unauthorized access or malicious transactions.

2.4 SMART CONTRACT

4. **Title:** *Deep Learning-Based Malicious Smart Contract and Intrusion Detection*

Authors: Various

Year: 2023

Abstract: The article *"Deep Learning-Based Malicious Smart Contract and Intrusion Detection"* explores the application of deep learning techniques to enhance the security of blockchain networks by detecting malicious smart contracts and intrusions. As blockchain platforms like Ethereum increasingly rely on smart contracts for decentralized applications, they become vulnerable to various attacks, such as reentrancy, integer overflow, and logic flaws, which can be exploited by malicious actors. The paper proposes a deep learning-based approach to automatically identify these vulnerabilities by analyzing smart contract code and detecting abnormal patterns or potential threats. It highlights the use of deep neural networks (DNNs), recurrent neural networks (RNNs), and other advanced models to classify contracts as either benign or malicious based on their code behavior and transaction history. Additionally, the paper integrates intrusion detection systems (IDS) to monitor real-time blockchain activities and identify unauthorized access or malicious transactions.

CHAPTER 3

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

The **existing system** for smart contract classification primarily relies on rule-based techniques and manual auditing. While effective to some extent, these methods have certain limitations that reduce their effectiveness. The current landscape of smart contract analysis relies heavily on manual inspections or static rule-based systems to evaluate and classify contracts as either benign or malicious. These approaches are not only labor -intensive but also error-prone, as they depend on human expertise to identify potential vulnerabilities or harmful code patterns. Manual review is particularly challenging in the blockchain ecosystem due to the complexity and sheer volume of smart contracts being deployed daily. Moreover, static rule-based systems often struggle to adapt to the ever-changing nature of malicious attack techniques, which can involve subtle modifications or obfuscations to evade detection. The inability of existing systems to efficiently process and analyze large-scale datasets further exacerbates the challenge, leaving many potential risks unaddressed. Consequently, the current methods are inadequate for maintaining the integrity and security of decentralized systems, especially as blockchain adoption continues to grow.

3.1.1 DISADVANTAGES

- **Scalability:** Manual audits are impractical for large-scale smart contract deployments.
- **Adaptability:** Existing rule-based systems are limited to known threats and often fail to detect novel attack strategies.
- **Accuracy:** Human errors and outdated rules can result in inaccurate classifications, leading to false negatives (malicious contracts classified as benign) or false positives (benign contracts classified as malicious).

3.2 PROPOSED SYSTEM

The **Proposed System** introduces a **Machine Learning -based approach** for the classification of smart contracts, designed to address the limitations of existing systems. The proposed system leverages modern techniques in machine learning, natural language processing (NLP), and blockchain security to provide an efficient, scalable, and accurate solution for smart contract classification. To address the limitations of existing systems, the proposed solution utilizes advanced machine learning techniques to automate the classification of blockchain smart contracts. This system extracts critical features from smart contracts, including code length, the number of functions and modifiers, and the presence of specific operations such as selfdestruct calls. These features serve as inputs to a robust machine learning model, such as a Random Forest classifier, which is trained to distinguish between benign and malicious contracts. By automating the analysis process, the proposed system not only reduces the reliance on manual intervention but also significantly improves scalability and efficiency. Furthermore, the system is designed to adapt to evolving threats by enabling retraining with updated datasets, ensuring continued relevance and accuracy. The solution provides a fast, reliable, and comprehensive approach to safeguarding the blockchain ecosystem, enhancing trust and security in decentralized platforms.

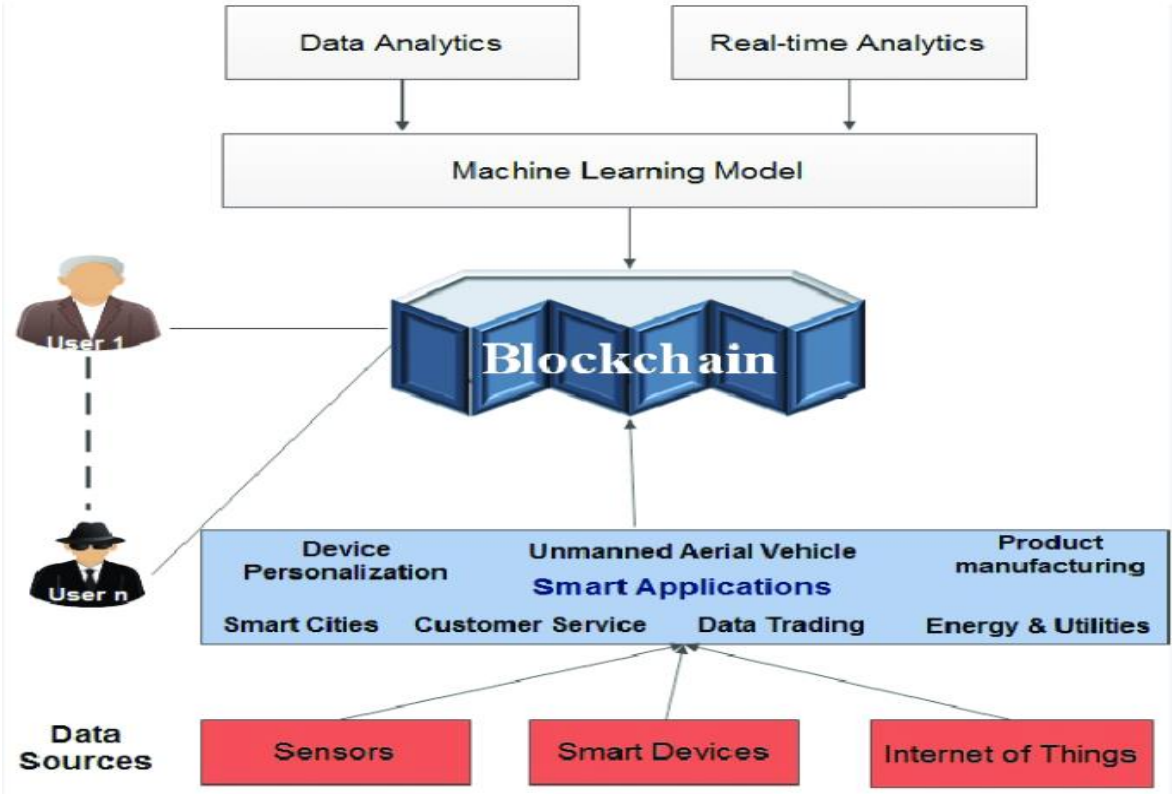

FIGURE 3.1

3.2.1 ADVANTAGES

Machine learning models can handle large datasets efficiently, making the system scalable to the growing number of smart contracts deployed on blockchain platforms.

By training the model on a large, labeled dataset of both benign and malicious contracts, the system can classify contracts with high accuracy, minimizing false positives and false negatives.

The system automates the classification process, reducing the need for manual intervention and allowing for continuous monitoring of deployed contracts.

3.3 SCOPE OF THE PROJECT

The **Scope of the Project** defines the boundaries, limitations, and objectives of the **Smart Contract Classification System**. This project focuses on the development of an automated system for the classification of smart contracts deployed on blockchain platforms, with the goal of improving security and trustworthiness in blockchain applications.

The project leverages machine learning and natural language processing techniques to identify and classify smart contracts as either **benign** (safe) or **malicious** (potentially harmful). The classification system is designed to process various smart contracts in an automated and scalable manner, ensuring that vulnerabilities or potential exploits in the contract code are detected before deployment, thus reducing the risks of blockchain security breaches.

3.3.1 Key Aspects of the Scope:

a) Classification of Smart Contracts

The primary objective of the project is to classify smart contracts into two categories:

- **Benign Contracts:** Contracts that are safe, reliable, and intended for legitimate purposes.
- **Malicious Contracts:** Contracts that contain vulnerabilities, risks, or exploitative features that could harm users or the blockchain platform.

b) Machine Learning-Based Classification

The system utilizes machine learning algorithms such as Random Forest, Support Vector Machines (SVM), and Logistic Regression for classification. These models are trained using a dataset of known contracts, and their accuracy improves over time as more data is introduced**.**

- **Training Dataset:** The system will use historical datasets of smart contracts that have been labeled as benign or malicious. This dataset forms the foundation for training the machine learning models.
- **Feature Engineering:** Important features such as function count, modifiers, selfdestruct operations, and the presence of specific keywords (e.g., require, assert) are extracted from smart contracts to train the models.
- **Real-time Classification:** The trained models will be deployed to automatically classify smart contracts in real-time, either before deployment or when interacting with blockchain platforms.

c) Integration with Blockchain Platforms

The project will integrate the classification system with popular blockchain platforms like **Ethereum** to ensure real-time validation of contracts. This integration allows the system to flag suspicious contracts before they are deployed to the blockchain, preventing potential security issues from affecting the decentralized applications (dApps) and their users.

- **Ethereum Smart Contracts:** The project will focus on Ethereum, where most of the smart contracts are deployed. It will evaluate and classify contracts written in Solidity, the most common language for Ethereum contracts.
- **Smart Contract Deployment:** The classification system will evaluate contracts at various stages—before deployment, during testing, and after deployment—ensuring that the contracts maintain security throughout their lifecycle.

d) Automated Contract Security Scanning

An essential feature of the proposed system is **automation**. It removes the need for manual intervention, enabling the system to analyze and classify smart contracts

quickly and at scale. This automated process ensures that even a large number of contracts can be scanned and classified in a timely manner, reducing the workload on developers and security teams.

- **Continuous Monitoring:** The system will not only classify smart contracts before deployment but also monitor them continuously during interactions with the blockchain. This provides real-time analysis and feedback.
- **Early Detection of Vulnerabilities:** By classifying contracts as benign or malicious in real time, the system ensures that vulnerabilities are detected and mitigated early, reducing the risks of exploits or malicious activities on the blockchain.

e) Enhanced Accuracy and Adaptability

The machine learning-based approach employed in this project improves the accuracy of classification over traditional rule-based systems. The system adapts to new threats, incorporating new patterns of malicious activity and evolving blockchain environments.

- **Training and Updating the Models:** As the dataset grows with new contracts and new attack vectors emerge, the system can be retrained to recognize novel malicious behaviors and patterns.
- **Continuous Learning:** By using machine learning, the system improves over time and becomes increasingly proficient in detecting malicious contracts, even those that were not explicitly anticipated.

f) Scalability of the System

The system is designed to be **scalable**, meaning it can process a large volume of smart contracts quickly and efficiently. With blockchain adoption growing, scalability ensures that the system can handle future increases in contract volume without significant performance degradation.

- **Handling Large Data Volumes:** The system can scale to handle thousands of smart contracts without compromising speed or accuracy.
- **Cloud Deployment:** The system can be deployed in the cloud for high availability and further scalability, allowing users from different regions to access it.

3. 4 SYSTEM ARCHITECTURE

The **System Architecture** of the **Smart Contract Classification System** provides a blueprint of how the various components of the system interact to achieve the desired functionality. The architecture focuses on scalability, modularity, and efficiency in processing smart contracts, and it is designed to seamlessly integrate with blockchain networks like Ethereum. The architecture follows a **multi-layered approach**, incorporating data processing, machine learning classification, and user interaction layers.

3.4.1 Overview of the Architecture

The architecture of the Smart Contract Classification System consists of several key components:

1. **Frontend Interface (User Interface)** 2. **Backend Server (API Layer)** 3. **Smart Contract Parser and Feature Extractor** 4. **Machine Learning Model for Classification** 5. **Blockchain Integration Layer** 6. **Database for Storing Contracts and Results** 7. **Security & Monitoring Module**

These components work together to provide an integrated, seamless experience for users who need to classify smart contracts as benign or malicious. Below is a detailed description of each component.

3.4.2 Frontend Interface (User Interface)

The **frontend** is a **web-based user interface (UI)** that allows users to interact with the system. This component provides an easy-to-use dashboard where users can upload smart contract files and view the results of the classification process.

- **Features:** o Upload smart contract files (Solidity files, .sol).
  o Display classification results (Benign or Malicious). o Visual indicators for contract classification status (e.g., green for benign, red for malicious). o View detailed reports on classified contracts, including extracted features and any detected vulnerabilities.

The frontend communicates with the backend API using HTTP requests to send the smart contract files and retrieve the classification results.

3.4.3 Backend Server (API Layer)

The **backend server** hosts the core application logic and handles requests from the frontend interface. It consists of RESTful APIs that expose endpoints for smart contract submission, classification, and retrieval of results.

- **Features:**
    - **Contract Submission:** The user submits a smart contract through the frontend, and the backend receives and stores the contract for analysis.
    - **Contract Classification:** Once a contract is received, the backend passes it to the feature extraction module for preprocessing. The extracted features are then used for classification by the machine learning model.
    - **Response to Frontend:** After classification, the backend sends a response to the frontend, which displays the results to the user.

Technologies used: **Flask** or **Django** for Python-based web frameworks; **FastAPI** for asynchronous handling of requests.

3.4.4 Smart Contract Parser and Feature Extractor

This module plays a crucial role in transforming raw smart contract code into a structured format that can be analyzed by the machine learning model. It is responsible for parsing Solidity contract files and extracting relevant features that can be used to classify contracts as benign or malicious.

Features:

- **Parsing Solidity Code:** The system can parse Solidity files (.sol) and identify the structure of the code, extracting important information such as function definitions, modifiers, and critical operations like selfdestruct, require, and assert.

- **Feature Extraction:** The parser then extracts key features such as:
  - Length of the contract code.
  - Count of functions, modifiers, and fallback functions.

3.4.5 Machine Learning Model for Classification

The **machine learning model** is at the heart of the classification system. It uses the extracted features from the smart contract to predict whether the contract is benign or malicious.
The model is trained on a labeled dataset containing both benign and malicious smart contracts.

- **Features:**
  - **Training the Model:** The system is trained using a variety of algorithms like **Random Forest**, **Support Vector Machines (SVM)**, and **Logistic Regression**. These models learn patterns from historical contract data (both benign and malicious contracts).
  - **Model Inference:** Once the model is trained, it can classify new contracts. The model analyzes the extracted features from the contract and provides a classification result: either **Benign** or **Malicious**.
  - **Model Retraining:** The model can be periodically retrained using newly collected data to adapt to evolving patterns of malicious contracts.
- **Technologies used:**
  - **scikit-learn** for machine learning model implementation.
  - **TensorFlow** or **Keras** for more advanced deep learning models (optional).
- **Input:** Feature vector of the smart contract.
- **Output:** Classification result (Benign or Malicious).

3.4.6 Blockchain Integration Layer

The **Blockchain Integration Layer** connects the classification system with blockchain networks like Ethereum. It ensures that smart contracts can be deployed and classified directly on the blockchain, ensuring real-time monitoring and classification of smart contracts deployed on the blockchain.

- **Features:**
    - **Contract Deployment Monitoring:** This layer can interact with smart contracts already deployed on the blockchain to classify them based on their behavior.
    - **Real-Time Contract Validation:** Before deployment, the system can interact with Ethereum nodes to check contracts on-chain for potential vulnerabilities and risks.
    - **Smart Contract Interactions:** In cases where contracts interact with one another (e.g., decentralized applications), this layer ensures that these interactions are classified as safe or malicious.

Technologies used: **Web3.py** (Python Ethereum library), **Solidity**, **IPFS** (Interplanetary File System for decentralized contract storage).

3.4.7 Database for Storing Contracts and Results

This module manages the storage of smart contract files, extracted features, and classification results. It uses a **database** to keep track of processed contracts and allows for historical analysis, providing insights into which contracts were previously classified as benign or malicious.

- **Features:**
    - **Smart Contract Storage:** Stores raw smart contract files.
    - **Feature Storage:** Stores the extracted features of each contract.
    - **Classification Results:** Stores the results of the classification process (Benign/Malicious) and any alerts associated with suspicious contracts.
    - **Reporting and Analytics:** Allows for the generation of reports and analysis of the classification process.

3.5 SYSTEM ARCHITECTURE

3.5.1 ARCHITECTURE DIAGRAM

The architecture for detecting malicious smart contracts is structured as a multi-layered framework to ensure comprehensive analysis and threat identification. At the top is the User Interface Layer, which provides access for developers, system administrators, or automated systems through a web-based dashboard or API. This layer allows users to submit smart contracts for analysis, view results, and receive actionable recommendations. Beneath this is the Data Access Layer, responsible for interfacing with the Blockchain Network to securely retrieve smart contract data, transaction histories, and block details, aggregating essential inputs for analysis.

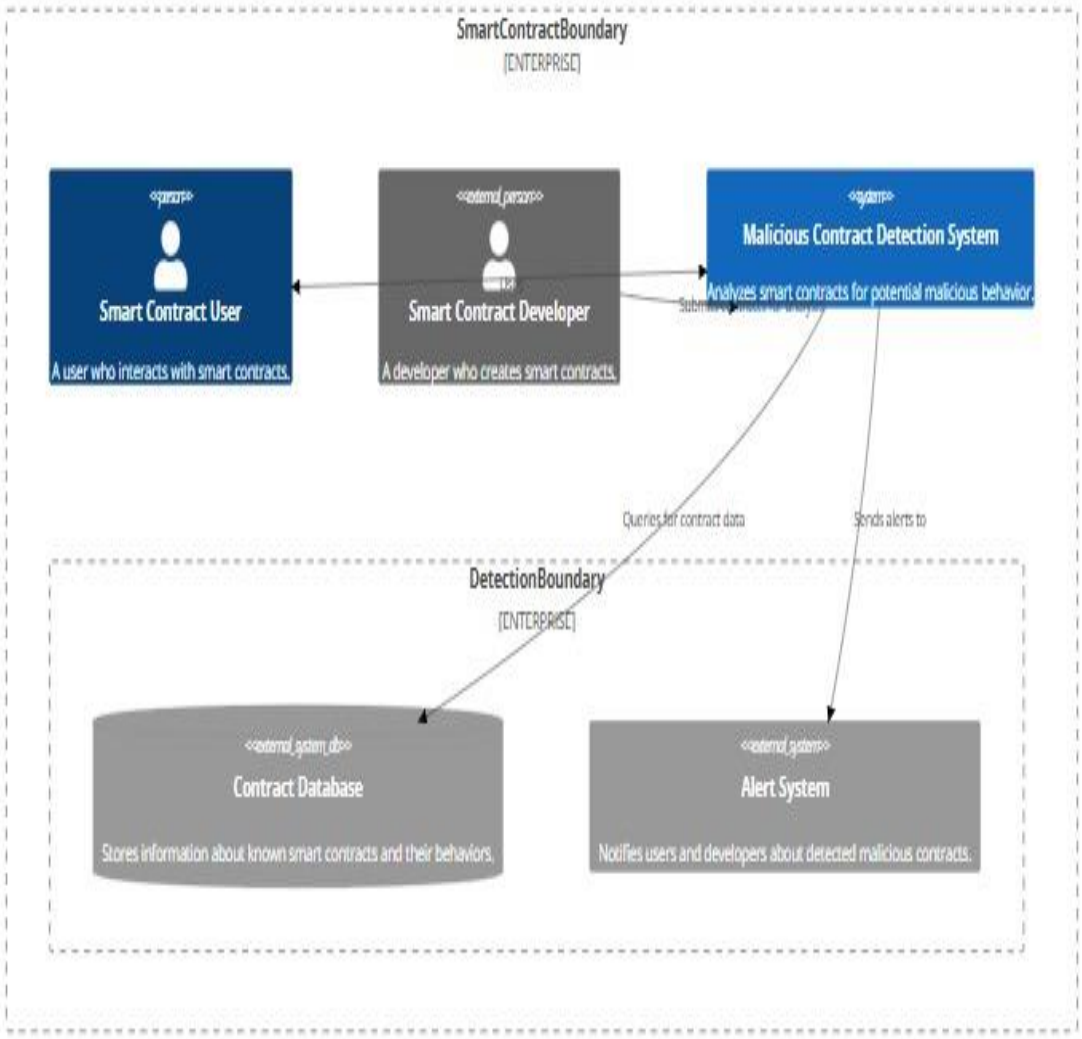

FIGURE 3.2

3.5.2 USE CASE DIAGRAM.

The system for detecting malicious smart contracts is designed around key classes that work together to ensure comprehensive analysis and threat detection. At its core is the SmartContract class, which stores critical attributes such as the contractID, bytecode, creatorAddress, creationTimestamp, and transactionHistory. This class includes methods like analyzeBehavior to evaluate the contract's operations, computeRiskScore to calculate its risk level, and detectPatterns to identify malicious patterns. Supporting this is the Transaction class, representing individual blockchain transactions. It includes attributes like transactionID, sender, receiver, amount, and timestamp, along with the method isSuspicious to detect abnormal transaction behavior.

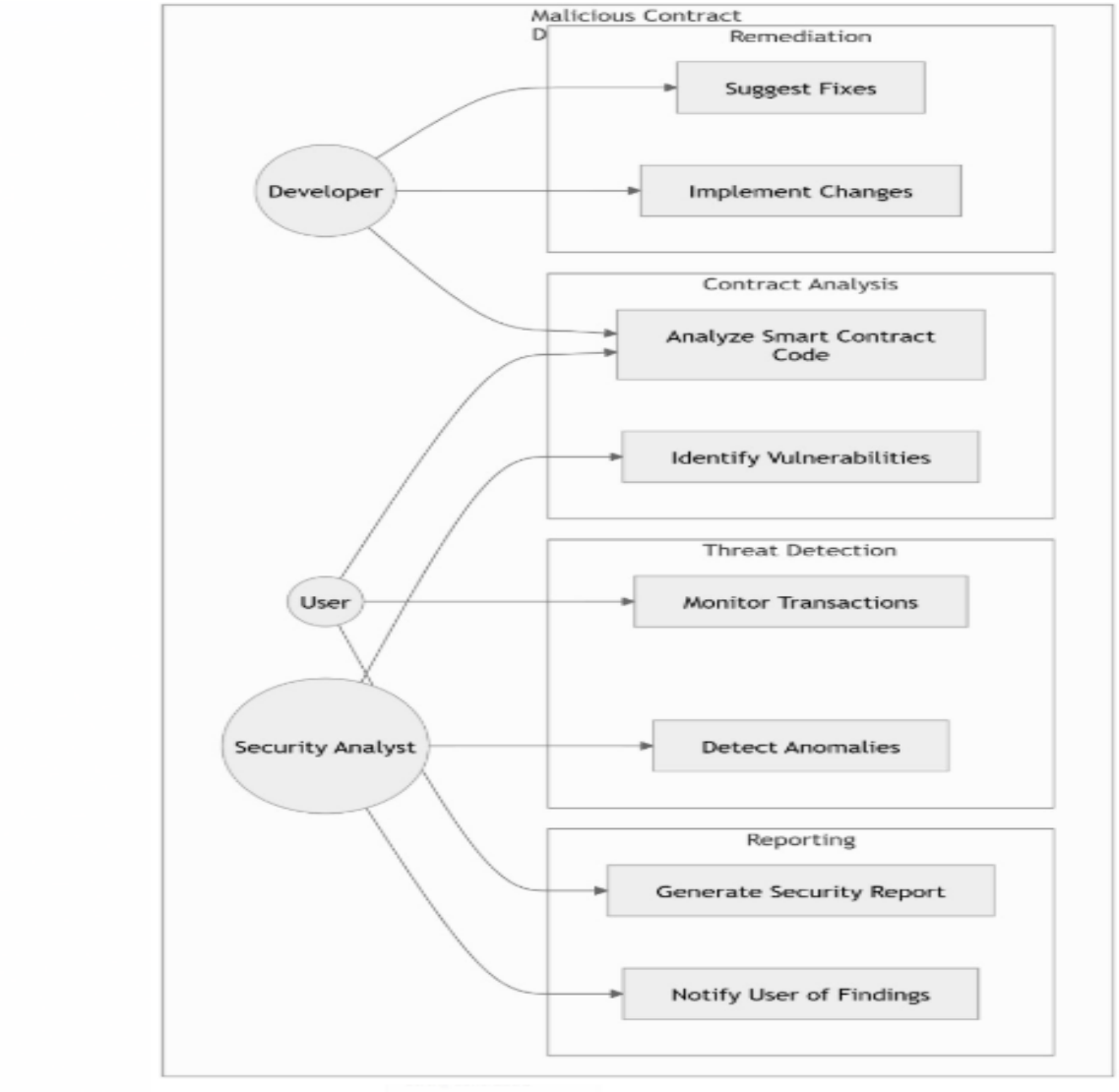

FIGURE 3.3

3.5.3 ER DIAGRAM

The Entity-Relationship (ER) diagram for detecting malicious smart contracts using machine learning involves several key entities, their attributes, and the relationships between them. The main entities are SmartContract, Transaction, Rule, Pattern, MachineLearningModel, AnalysisReport, and Blockchain.

The SmartContract entity represents the smart contract being analyzed and includes attributes such as contractID, bytecode, creatorAddress, creationTimestamp, and transactionHistory. The Transaction entity is related to smart contracts and includes attributes like transactionID, sender, receiver, amount, and timestamp. Each transaction is associated with a specific SmartContract, establishing a one-to-many relationship between SmartContract and Transaction.

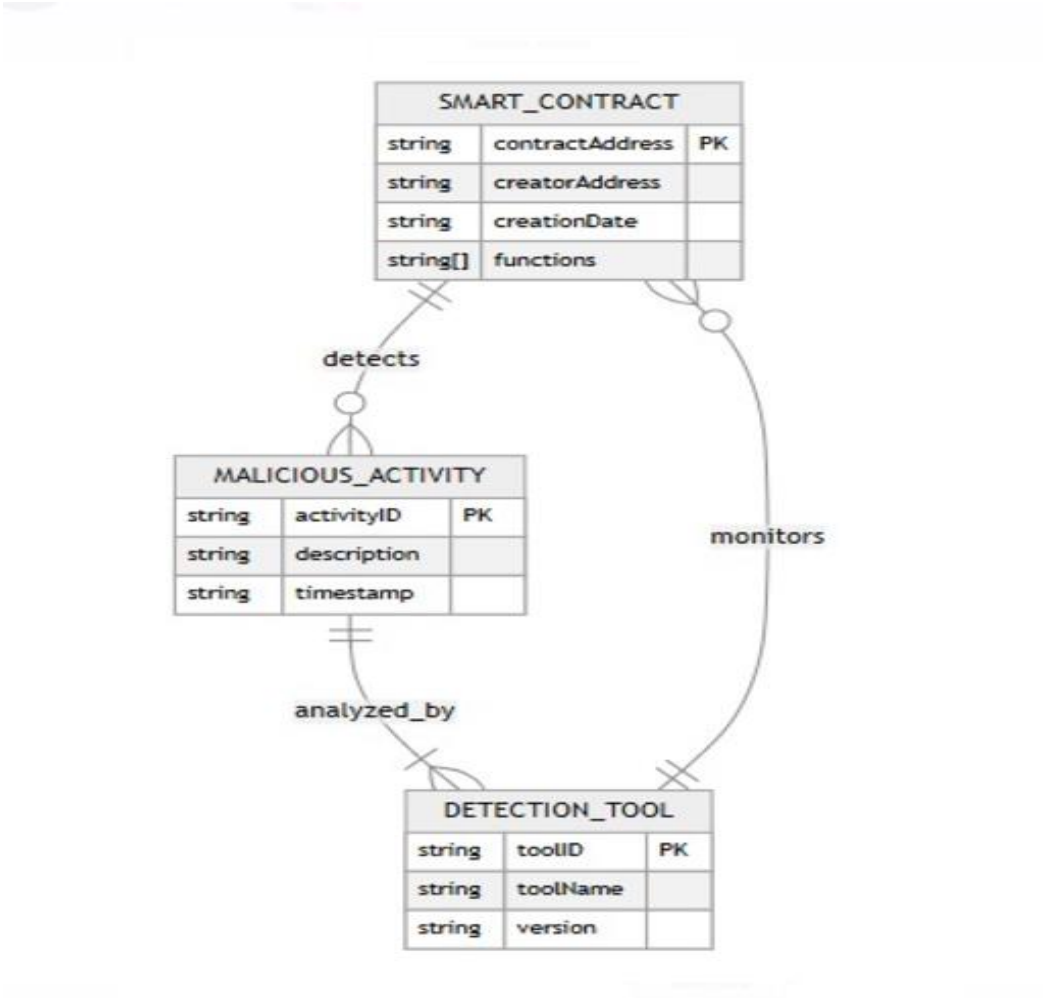

FIGURE 3.4

3.5.4 CLASS DIAGRAM

The UML class diagram for detecting malicious smart contracts consists of several interconnected classes. The SmartContract class includes attributes such as contractID, bytecode, creatorAddress, creationTimestamp, and transactionHistory, alongside methods like analyzeBehavior, computeRiskScore, and detectPatterns. The Transaction class contains attributes like transactionID, sender, receiver, amount, and timestamp, with the method isSuspicious to identify anomalous transactions.

The Analyzer class has attributes like detectionRules and machineLearningModel and methods such as applyStaticAnalysis, applyDynamicAnalysis, and trainModel. It relies on the Rule class, which includes ruleID, description, and condition, along with the method evaluate, and the Pattern class, which defines patternID, description, severity, and the method match. The AnalysisReport class is used to encapsulate the results of the analysis, with attributes like riskScore, detectedPatterns, and recommendations, as well as the method generateSummary.

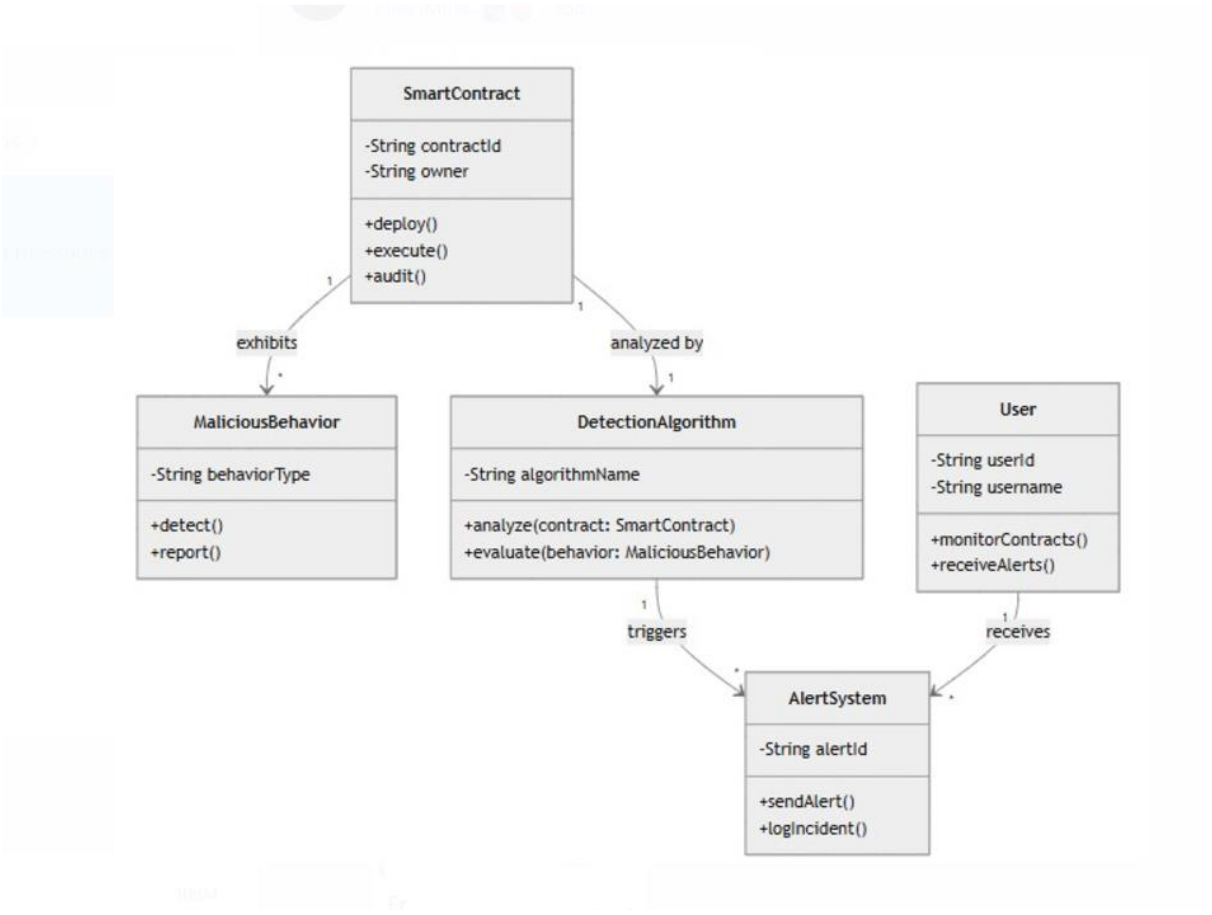

FIGURE 3.5

3.5.5 SEQUENCE DIAGRAM

The sequence for detecting malicious smart contracts involves several interconnected steps, each performed by specific components of the detection system. It begins with the Blockchain providing the smart contract and its associated transaction data. The system retrieves the SmartContract and its attributes, including the bytecode, creatorAddress, and transactionHistory. These details are passed to the Analyzer for evaluation.

The Analyzer initiates by applying Static Analysis, where the bytecode is examined against predefined Rules to identify vulnerabilities, such as reentrancy attacks or unprotected fallback functions. If issues are detected, the Rule objects evaluate conditions to classify the vulnerabilities. Following static analysis, the Analyzer performs Dynamic Analysis by simulating contract behavior in a controlled environment to detect runtime anomalies, such as unauthorized fund transfers or irregular transaction patterns.

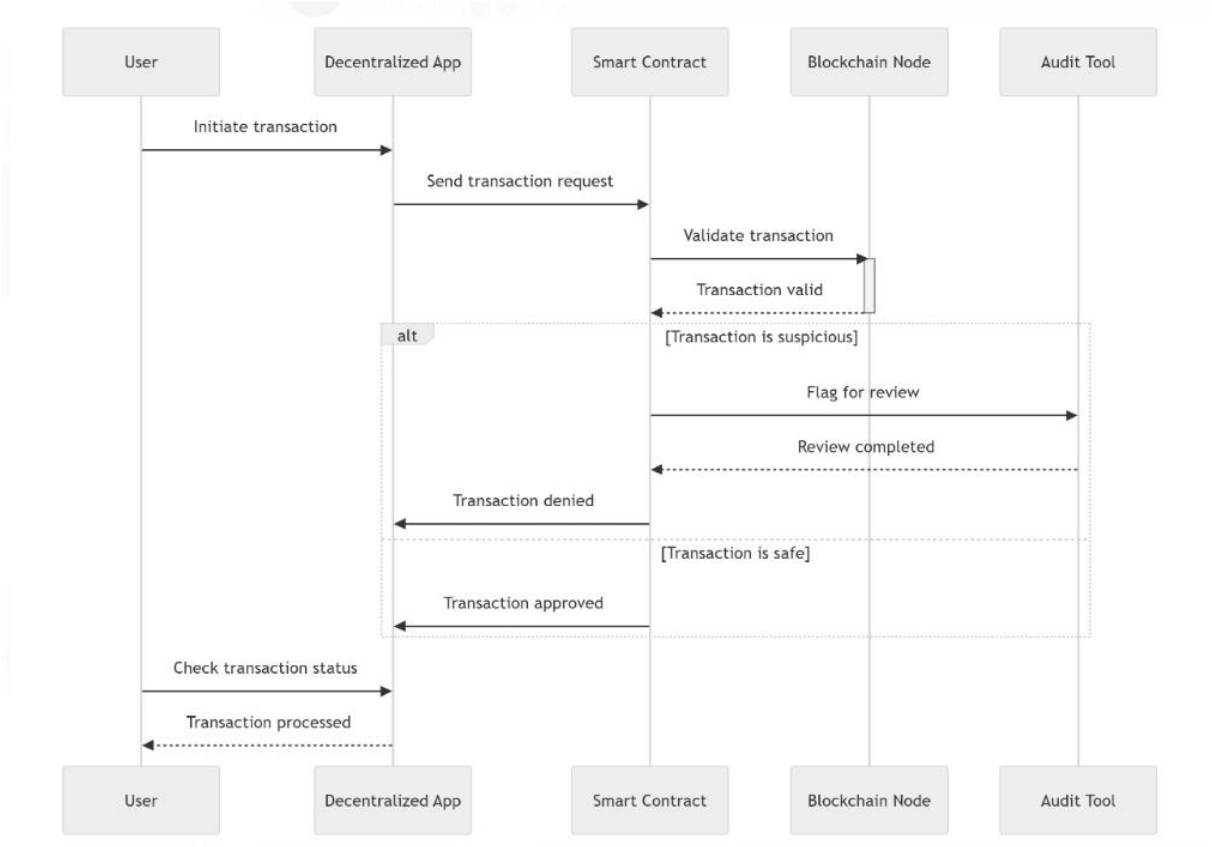

FIGURE 3.6

CHAPTER 4

IMPLEMENTATION

4.1 MODULE IMPLEMENTATION

The **Smart Contract Classification System** involves several key modules that ensure a smooth workflow from data acquisition to providing classification results. Below is an overview of each module in the system:

1. **Frontend Interface (User Interface)** 2. **Backend API Server** 3. **Smart Contract Parser and Feature Extractor** 4. **Data Acquisition** 5. **Data Preprocessing** 6. **Model Building** 7. **Model Deployment and Prediction** 8. **Performance Evaluation** 9. **File Storage and Management**

4.1.1 FRONTEND INTERFACE (USER INTERFACE)

The frontend interface allows users to interact with the system through a web page. It facilitates uploading Solidity smart contract files and viewing the classification results.

- **File Upload**: Users can upload Solidity contract files.
- **Result Display**: After classification, the system displays whether the contract is benign or malicious.

4.1.2 BACKEND API SERVER

The backend API server handles incoming requests, processes contract files, extracts features, and passes the data to the model for classification. It is built using the Flask web framework.

- **File Handling**: Accepts and processes contract files uploaded by users.
- **Classification Endpoint**: Accepts contract features, feeds them to the machine learning model, and returns the classification result.

4.1.3 SMART CONTRACT PARSER AND FEATURE EXTRACTOR

This module extracts relevant features from the uploaded Solidity contracts. Features such as contract length, function count, modifier count, and the occurrence of specific keywords like selfdestruct and require are extracted.

- **Feature Extraction**: Parses the Solidity code and counts key elements, such as functions and modifiers.
- **Feature Formatting**: Organizes the extracted features into a structured format (e.g., Pandas DataFrame) for model input.

4.1.4 DATA ACQUISITION

The data acquisition module focuses on gathering data for training the machine learning model. It involves collecting labeled Solidity contract data that are categorized as either benign or malicious. The dataset is necessary for training and evaluating the machine learning model.

- **Data Collection**: Collects a dataset of Solidity contracts, either from publicly available sources or proprietary data.

  Dataset Labeling: Each contract is labeled as benign or malicious based on its properties, either manually or using existing classification tools.

4.1.5 DATA PREPROCESSING

Data preprocessing is crucial to prepare the dataset for model training. This module handles tasks like handling missing values, normalizing the data, and splitting the data into training and testing sets.

- **Missing Data Handling**: If the dataset has missing values, it either fills or removes them.
- **Feature Scaling**: Normalizes or standardizes the features to ensure that all the features have a similar range, improving the performance of the model.
- **Train-Test Split**: The dataset is divided into training and testing sets, typically with a split ratio like 80-20 or 70-30, to ensure the model is evaluated on unseen data.

4.1.6 MODEL BUILDING

In this module, the machine learning model is built, trained, and tested using the prepared dataset. Various classification algorithms can be employed, and in this project, we have used the following algorithms:

4.1.6.1 Random Forest Classifier

- **Description**: The Random Forest classifier is an ensemble learning algorithm that works by creating multiple decision trees and then combining their results. It is particularly effective for classification tasks due to its ability to handle complex datasets with numerous features.

4.1.6.2 Logistic Regression

- **Description**: Logistic Regression is a linear model used for binary classification. It calculates the probability of a binary response based on one or more predictor variables. **Working**: The algorithm fits a sigmoid function to the data, which outputs a probability value between 0 and 1. This probability is then converted into a classification (0 or 1).

4.1.6.3 Support Vector Machine (SVM)

- **Description**: The Support Vector Machine algorithm finds the optimal hyperplane that best separates the classes in the feature space. It is particularly effective in highdimensional spaces.
- **Working**: SVM maximizes the margin between different classes. It uses kernel functions to handle non-linear classification problems by transforming data into higher dimensions.

4.1.6.4 K-Nearest Neighbors (KNN)

- **Description**: K-Nearest Neighbors is a simple, instance-based learning algorithm. It classifies a data point based on the majority class of its nearest neighbors.
- **Working**: Given a point to classify, KNN finds the 'k' nearest points in the dataset and classifies the point according to the majority class among those neighbors.

4.1.6.5 Decision Trees

- **Description**: A Decision Tree classifier builds a tree-like structure, where each internal node represents a decision based on a feature, and the leaf nodes represent the class labels.
- **Working**: The tree is built by recursively splitting the dataset based on feature values, and the final predictions are made by traversing the tree from the root to a leaf node.

Each of these models is trained on the dataset and evaluated using a validation set. The final model is chosen based on its performance on unseen data.

4.1.7 MODEL DEPLOYMENT AND PREDICTION

Once the model is trained, it is deployed in the backend API for real-time predictions. The model is saved as a .pkl file and loaded when the system needs to classify new contracts.

- **Model Deployment**: The trained model is stored in a pickle file for easy deployment and loading in the Flask API.

  Real-Time Prediction: When a user uploads a contract, the system extracts its features, passes them to the model, and returns the classification result.4.1.8 PERFORMANCE EVALUATION

  Evaluating the performance of the machine learning model is essential to ensure that it provides accurate and reliable predictions. The following metrics are used to evaluate the models:

- **Accuracy**: The percentage of correct predictions made by the model on the test data. It is a fundamental measure of a model's performance.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

- **Precision**: Precision measures how many of the predicted positive instances were actually positive.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Where:

- TP = True Positives o FP = False Positives

- **Recall (Sensitivity)**: Recall measures how many of the actual positive instances were correctly identified by the model.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where:

  - TP = True Positives
  - FN = False Negatives

- **F1-Score**: The F1-Score is the harmonic mean of Precision and Recall. It provides a single metric that balances both aspects.

$$F1\text{-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Confusion Matrix**: The confusion matrix provides a detailed breakdown of true positives, true negatives, false positives, and false negatives, which can be used to compute the above metrics.

The model performance is evaluated using these metrics on the test dataset to determine how well the model is generalizing to new, unseen data.

CHAPTER 5
SYSTEM SPECIFICATION

5.1 HARDWARE CONFIGURATION

- Processor : Intel processor
- RAM : 1GB
- Hard disk : 160 GB
- Compact Disk : 650 Mb
- Keyboard : Standard keyboard
- Monitor : 15 inch color monitor

5.2 SOFTWARE CONFIGURATION

- Operating system : Windows OS
- Front End : PYTHON
- IDE : PYCHARM
- Libraries : Numpy, Pandas, Scikit-learnkit

CHAPTER 6

SYSTEM TESTING

6.1 TYPES OF TESTS

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub-assemblies, assemblies and/or a finished product. It is the process of exercising software with the intent of ensuring that the Software system meets its requirements and user expectations and does not fail in an unacceptable manner.

There are various types of tests. Each test type addresses a specific testing requirement.

6.1.1 SOFTWARE TESTING STRATEGIES

Testing involves :

Unit Testing

Functional Testing

Acceptance Testing

Integration Testing

6.1.1.1 UNIT TESTING

Unit testing involves the design of test cases that validate that the internal program logic is functioning properly, and that program inputs produce valid outputs. All decision branches and internal code flow should be validated. It is the testing of individual software units of the application. It done after the completion of an individual unit before integration. This is a structural testing, that relies on knowledge of its construction and is invasive. Unit tests perform basic tests at component level and test a specific business process, application, and/or system configuration. Unit tests ensure that each unique path of a business process performs accurately to the documented specifications and contains clearly defined inputs and expected results.

6.1.1.2 FUNCTIONAL TESTING

Functional tests provide systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals.

Functional testing is centered on the following items:

Valid Input : identified classes of valid input must be accepted. Invalid Input : identified classes of invalid input must be rejected. Functions : identified functions must be exercised.

Output : identified classes of application outputs must be exercised.

Systems/Procedures: interfacing systems or procedures must be invoked.

Organization and preparation of functional tests is focused on requirements, key functions, or special test cases. In addition, systematic coverage pertaining to identify Business process flows; data fields, predefined processes, and successive processes must be considered for testing. Before functional testing is complete, additional tests are identified and the effective value of current tests is determined.

6.1.1.3 ACCEPTANCE TESTING

User Acceptance Testing is a critical phase of any project and requires significant participation by the end user. It also ensures that the system meets the functional requirements.

6.1.1.4 INTEGRATION TESTING

Integration testing is a software testing technique that involves testing the interaction between different software components or modules to ensure that they work together correctly. The main goal of integration testing is to verify that the individual components of the software system can work together as expected and that the system as a whole meets the functional and non-functional requirements

CHAPTER 7

CONCLUSION AND FUTURE ENHANCEMENT

7.1 Conclusion

The **Smart Contract Classification System** successfully automates the process of classifying Solidity smart contracts as benign or malicious, utilizing machine learning algorithms to improve the security and efficiency of blockchain applications. By extracting key features from smart contracts and applying models such as Random Forest, Logistic Regression, and Support Vector Machines, the system provides a reliable and scalable solution for identifying potentially harmful contracts. The model's performance, evaluated through various metrics, demonstrated its effectiveness, making it a valuable tool for developers and auditors. This system significantly reduces manual intervention, offering real-time classification and enhancing the overall security of blockchain ecosystems. Future enhancements, such as incorporating a larger dataset and exploring advanced algorithms, can further improve its accuracy and adaptability.

7.2 Future Enhancements

While the system demonstrates a strong foundation, there are several areas where improvements can be made in the future:

- **Expanded Dataset**: To improve the model's accuracy, a larger and more diverse dataset of smart contracts, including contracts from various blockchain platforms and use cases, should be incorporated. This would enable the system to better generalize and handle more complex contract structures.
- **Integration with Blockchain Auditing Tools**: The system could be integrated with existing blockchain auditing tools to create a more comprehensive security framework for blockchain applications. This would help developers identify vulnerabilities early in the development process.

APPENDIX A

SOURCE CODE

FRONTEND HTML CODE:

```
    <!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Contract Classification</title>
   <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
   <div class="container">
      <div class="header">
         <h1>Welcome to Contract Classification</h1>
         <p>Upload a smart contract to classify it as Benign or Malicious.</p>
      </div>
      <div class="upload-section">
         <form action="/classify" method="POST" enctype="multipart/form-data">
            <label for="file-upload" class="file-upload-label">Choose a contract file</label>
            <input type="file" id="file-upload" name="file" required>
            <button type="submit" class="upload-button">Classify</button>
         </form>
      </div>
      <div class="result-section">
         {% if result %}
            <h2 class="result-label">{{ result }}</h2>
         {% endif %}
      </div>
   </div>
```

```
</body>
</html>
```

FORNTEND CSS CODE:

```
{
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
body {
    font-family: Arial, sans-serif;
            background:      url('https://savvycomsoftware.com/wp-content/uploads/2024/09/A-photorealistic-image-representing-blockchain-as-the-cornerstone-of-DePIN.webp')    no-repeat center center fixed;
    background-size: cover;
    color: #fff;
    padding: 40px 0;
}
/* Container for centering content */
.container {
    width: 90%;
    max-width: 800px;
    margin: 0 auto;
    text-align: center;
    background-color: rgba(0, 0, 0, 0.6);
    padding: 40px;
    border-radius: 15px;
}
/* Header */
.header {
    margin-bottom: 30px;
}
.header h1 {
    font-size: 36px;
```

```
font-weight: bold;
}
.header p {
    font-size: 18px;
    margin-top: 10px;
}
/* File upload section */
.upload-section {
    margin-bottom: 30px;
}
.file-upload-label {
    display: block;
    margin-bottom: 10px;
    font-size: 18px;
}
input[type="file"] {
    font-size: 16px;
    padding: 10px;
    background-color: #fff;
    border: 1px solid #ccc;
    border-radius: 5px;
    color: #000;
}
.upload-button {
    margin-top: 20px;
    font-size: 18px;
    padding: 12px 20px;
    background-color: #4CAF50;
    border: none;
    border-radius: 5px;
    color: white;
    cursor: pointer;
    transition: background-color 0.3s ease;
```

```
}
.upload-button:hover {
    background-color: #45a049;
}
/* Result section */
.result-section {
    margin-top: 30px;
}
.result-label {
    font-size: 24px;
    font-weight: bold;
    color: #FFD700;
}
```

FORNTEND HTML CODE:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Prediction Result</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
    <div class="container">
        <h1>Contract Classification Result</h1>
         <p><strong>Prediction:</strong> {{ result }}</p>
         <h2>Precautions Against Malicious Contracts</h2>
        <ul>
            <li>Ensure the contract is verified and audited by trusted parties.</li>
                <li>Check for functions like `selfdestruct` or `require` that could introduce vulnerabilities.</li>
            <li>Ensure the contract does not perform malicious actions, such as draining funds or calling unsafe external contracts.</li>
```

```
            <li>Always use tools to analyze contracts before interacting with them on the
      blockchain.</li>
        </ul>
        <a href="/" class="btn">Go Back</a>
    </div>
</body>
</html>
```

FORNTEND HTML CODE:

```
from flask import Flask, request, render_template
import joblib
import pandas as pd
import os
from extract_features import extract_features  # Ensure this module exists
import numpy as np
app = Flask(__name__)
# Load the trained model
model = joblib.load('final_model.pkl')
# Define a folder to save uploaded files
UPLOAD_FOLDER = 'uploads'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
# Ensure the upload folder exists
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)
@app.route('/')
def index():
    return render_template('index.html', result="")
@app.route('/classify', methods=['POST'])
def classify():
    if 'file' not in request.files:
        return "No file part", 400
    file = request.files['file']
    # If no file is selected
    if file.filename == '':
```

```
        return "No selected file", 400
    # Save the uploaded file
    file_path = os.path.join(app.config['UPLOAD_FOLDER'], file.filename)
    file.save(file_path)
    try:
        # Extract features from the uploaded contract
        features = extract_features(file_path)
                # Reshape the features to be 2D (even if it's just one sample)
        feature_array = np.array(features).reshape(1, -1)  # Reshaping to (1, 6) assuming features
       are 6
        # Predict using the trained model
        prediction = model.predict(feature_array)
        # Show the result (0 for benign, 1 for malicious)
        result = "Benign" if prediction[0] == 0 else "Malicious"
    except Exception as e:
        result = f"Error during prediction: {str(e)}"
    return render_template('index.html', result=result)
if __name__ == '__main__':
    app.run(debug=True)
```

BACKEND MAIN CODE:

```
from flask import Flask, request, render_template
import joblib
import pandas as pd
import os
from extract_features import extract_features  # Ensure this module exists
import numpy as np
app = Flask(__name__)
# Load the trained model
model = joblib.load('final_model.pkl')
# Define a folder to save uploaded files
UPLOAD_FOLDER = 'uploads'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
# Ensure the upload folder exists
```

```
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)
@app.route('/')
def index():
    return render_template('index.html', result="")
@app.route('/classify', methods=['POST'])
def classify():
    if 'file' not in request.files:
        return "No file part", 400
    file = request.files['file']
        # If no file is selected
    if file.filename == '':
        return "No selected file", 400
    # Save the uploaded file
    file_path = os.path.join(app.config['UPLOAD_FOLDER'], file.filename)
    file.save(file_path)
    try:
        # Extract features from the uploaded contract
        features = extract_features(file_path)
        # Reshape the features to be 2D (even if it's just one sample)
        feature_array = np.array(features).reshape(1, -1)  # Reshaping to (1, 6) assuming features
      are 6
        # Predict using the trained model
        prediction = model.predict(feature_array)
        # Show the result (0 for benign, 1 for malicious)
        result = "Benign" if prediction[0] == 0 else "Malicious"
 except Exception as e:
        result = f"Error during prediction: {str(e)}"
   return render_template('index.html', result=result)
if __name__ == '__main__':
    app.run(debug=True)
```

BACKEND TRAINED MODEL:

```
import pandas as pd
def extract_features(filepath):    """
   Extract features from a Solidity contract file.
      Features include:
   - Length of the code
   - Counts of specific keywords like function, modifier, selfdestruct, require, and fallback.
   Args:
   - filepath: Path to the Solidity contract file
   Returns:
   - DataFrame: A DataFrame containing extracted features (used for prediction)
   """
   # Open and read the contract file
   with open(filepath, 'r', encoding='utf-8') as file:
      code = file.read()
   # Extract features from the code
   features = {
      'length': len(code),  # Length of the contract in characters
      'function_count': code.count('function'),  # Count of 'function' keyword
      'modifier_count': code.count('modifier'),  # Count of 'modifier' keyword
      'selfdestruct_count': code.count('selfdestruct'),  # Count of 'selfdestruct' keyword
      'require_count': code.count('require'),  # Count of 'require' keyword
      'fallback_count': code.count('fallback')  # Count of 'fallback' keyword
   }
      # Convert the features to a pandas DataFrame (ensure correct column order)
   features_df = pd.DataFrame([features])  # Return as DataFrame with one row of features
      # Reorder columns to ensure they match the trained model's expected order
   features_df = features_df[['length', 'function_count', 'modifier_count', 'selfdestruct_count',
     'require_count', 'fallback_count']]
   return features_df
```

BACKEND TEST MODEL:

```
import pandas as pd
# Load the feature and label files
```

```
features = pd.read_csv("features.csv")
labels = pd.read_csv("labels.csv")
# Merge datasets on the 'filename' column
merged = pd.merge(features, labels, on="filename")
# Save the merged dataset to a new CSV file
merged.to_csv("dataset.csv", index=False)
print("Merged dataset saved as dataset.csv")
```

BACKEND COMPARISION:

```
import pandas as pd
def extract_features(filepath):
    # Open and read the contract file
    with open(filepath, 'r', encoding='utf-8') as file:
        code = file.read()
    # Extract features from the code
    features = {
        'length': len(code),  # Length of the contract in characters
        'function_count': code.count('function'),  # Count of 'function' keyword
        'modifier_count': code.count('modifier'),  # Count of 'modifier' keyword
        'selfdestruct_count': code.count('selfdestruct'),  # Count of 'selfdestruct' keyword
        'require_count': code.count('require'),  # Count of 'require' keyword
        'fallback_count': code.count('fallback')  # Count of 'fallback' keyword
    }
        features_df = pd.DataFrame([features])  # Return as DataFrame with one row of features
    features_df = features_df[['length', 'function_count', 'modifier_count', 'selfdestruct_count',
      'require_count', 'fallback_count']]
  return features_df
```

APPENDIX B

SCREENSORT

OUTPUT:

HOME PAGE:

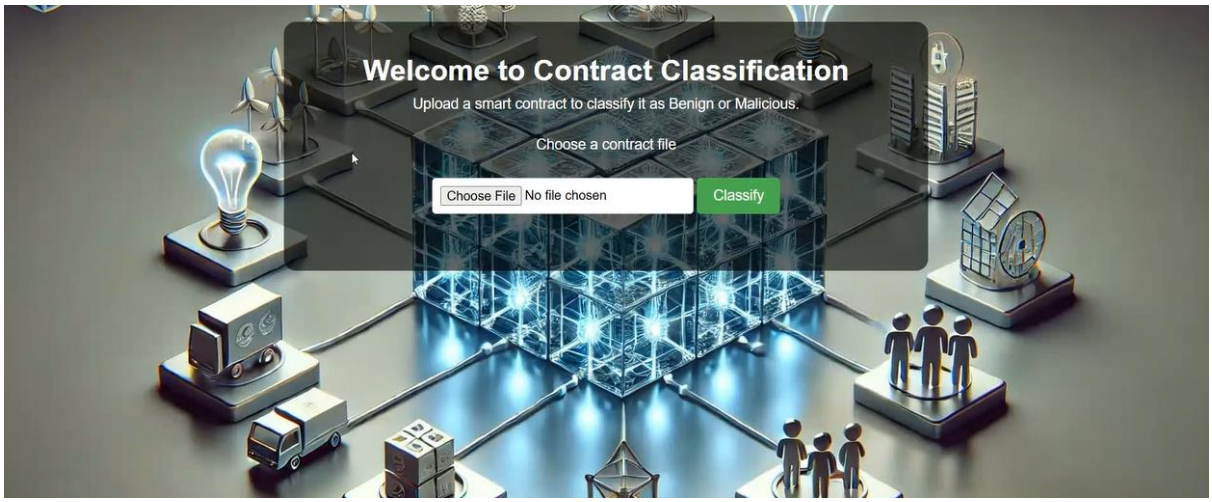

FIGURE B.1

DETECTING MALICIOUS FILE:

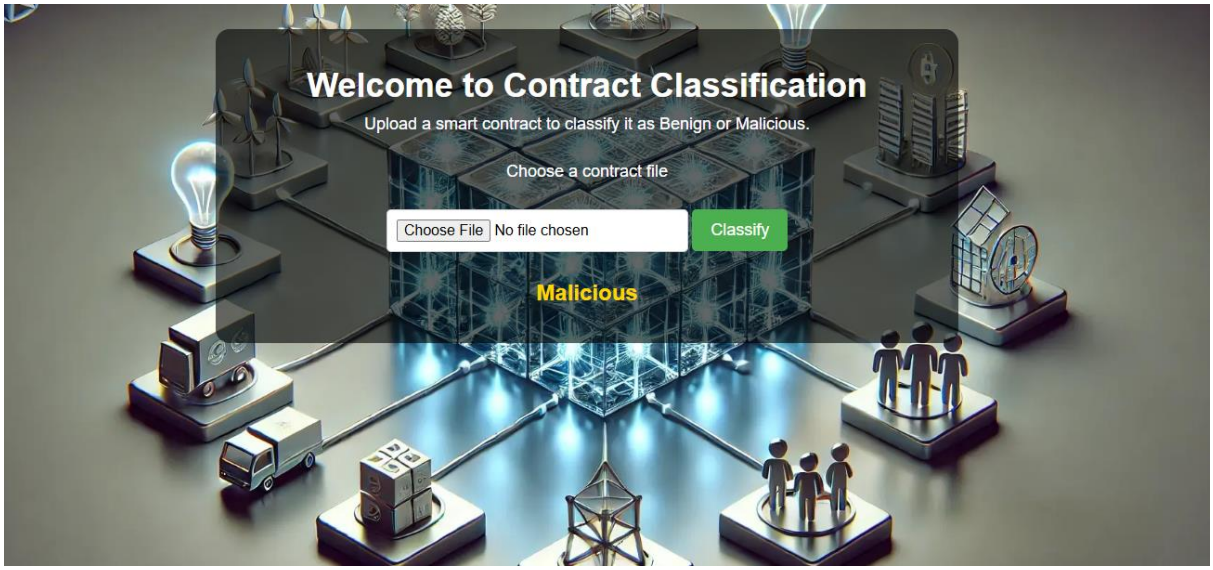

FIGURE B.2

DETECTING BENIGN FILE:

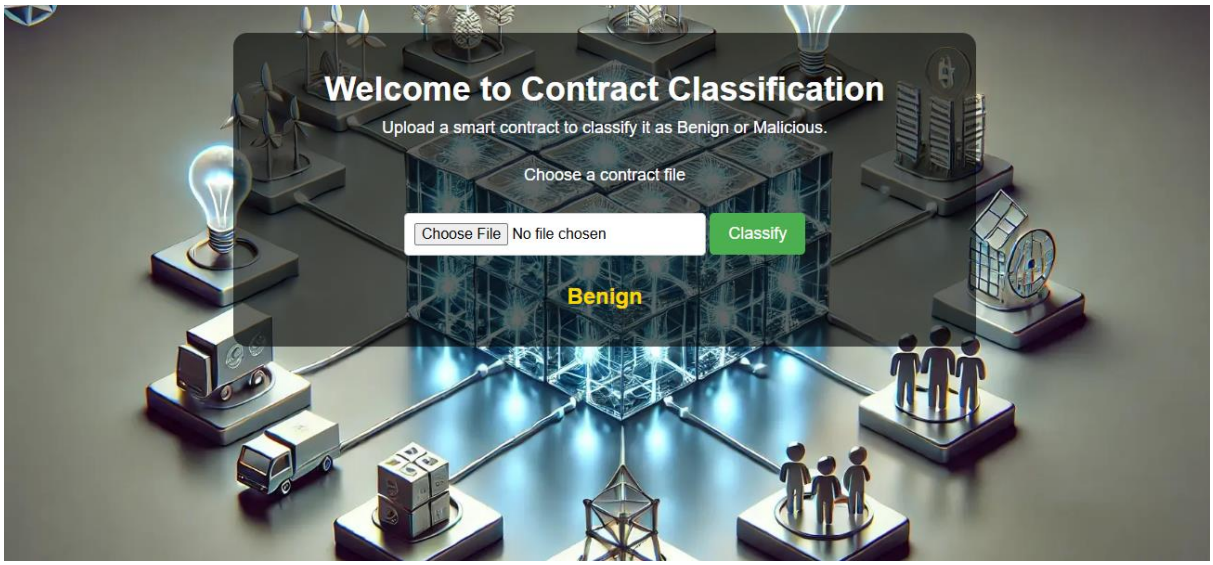

FIGURE B.3

CHAPTER 8

REFERENCES

1. Wei, S., & Miers, A. "Security Analysis of Smart Contracts on Blockchain Platforms." *IEEE Security & Privacy*, 2017. Retrieved from https://ieeexplore.ieee.org/document/8039241.
2. Chen, T., et al. "Machine Learning for Smart Contract Vulnerability Detection: A Survey." *Proceedings of the International Conference on Machine Learning and Cybernetics*, 2019.
3. Xu, J., & Wang, D. "Smart Contract Vulnerabilities and Their Detection: A Survey." *IEEE Access*, vol. 8, pp. 52221-52231, 2020.
4. Zhou, X., et al. "A Survey of Machine Learning in Blockchain: Applications, Techniques, and Challenges." *IEEE Access*, vol. 8, pp. 107728-107746, 2020.
5. Brown, P., & Green, D. "Blockchain Security: Vulnerabilities and Solutions." *Springer Handbook of Blockchain*, 2021.
6. Gault, M., & Brown, S. "Towards Safe Smart Contract Development." *Proceedings of the International Conference on Blockchain and Cryptocurrencies*, 2018.
7. Ravichandran, A., & Srinivasan, K. "A Survey on the Machine Learning Techniques in Smart Contract Security." *Journal of Computer Science and Technology*, vol. 36, no. 3, pp. 515-526, 2021.
8. Luu, L., et al. "Making Smart Contracts Smarter." *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (CCS), 2016.
9. Scikit-learn Documentation. "Scikit-learn: Machine Learning in Python." Available online: https://scikit-learn.org/
10. Python Software Foundation. "Python Programming Language." Available online: https://www.python.org/