”Streamlined Mechanism for Smart Contract Security: Mitigating Vulnerabilities and Enhancing Trust in Blockchain Technology”

***Abstract*—Blockchain stands as a transformative technology, fostering trustless communication by allowing users to securely store data across a multitude of computers in an immutable format. This capability extends to the deployment of smart contracts—compact programs enabling automatic enforcement of agreed-upon terms between untrusted parties. However, the Ethereum blockchain, a prominent platform for smart contracts, is not immune to security vulnerabilities, potentially leading to unintended behaviors and significant financial losses given the substantial cryptocurrency values involved. Current challenges in smart contract security have hindered the widespread practicality of blockchain applications. Existing research predominantly focuses on detecting vulnerabilities off-chain, yet once a smart contract is deployed on-chain, it becomes immutable. Consequently, these deployed contracts remain susceptible to attacks, necessitating innovative solutions. In response to this issue, we introduce a novel smart contract interface known as the Streamlined Interface for Detection of Vulnerabilities and Deployment of Smart Contracts. This interface offers automated patching for vulnerable deployed contracts without altering the contract codes. The core concept behind the Streamlined Interface for Detection of Vulnerabilities and Deployment of Smart Contracts involves generating patch contracts to proactively prevent malicious transactions. Specifically addressing critical bug types such as reentrancy, arithmetic bugs, and unchecked low-level checks, we demonstrate how this interface autonomously rectifies these issues on-chain. In a practical evaluation, the model successfully identified 19 out of 20 known vulnerabilities present in the provided set of smart contracts, achieving an impressive 95% accuracy. This innovative approach marks a significant stride toward enhancing the security and reliability of blockchain-based smart contracts.**

***Index Terms*—Ethereum Virtual Machine (EVM), GETH, Smart Contracts, Vulnerabilities, Machine Learning, Blockchain**

I. Introduction

Blockchain technology (BC) in recent years, has revolutionize a variety of industries such as healthcare [1], financial, insurance, government, real estate, social media, supply chain, and education,etc. It efficiently and logically answers a number of interesting real-world challenges. The term BC denotes a significant shift in how data is acquired, stored, and delivered due to 3 key characteristics: P2P network, decentralization, and immutability. Whereas the 1st generation of BC technology is almost purely used for cryptocurrencies like Bitcoin [2].The 2nd generation, demonstrated by Ethereum [3], is an open and decentralized platform that enables a new computing paradigm. These complex system's semantic eventually introduce dozens of new security flaws that don't exist in pure monetary systems like Bitcoin. Ethereum is a widely preferred trade platform that allows an account holder to perform exchange, buy, and sell cryptocurrency with the help of smart contracts, these contracts automatically executed when preset terms and conditions are satisfied and are maintained in BC. Contracts have the unique ability to transfer ether to and from users as well as other contracts. Users submit transactions to the Ethereum network for creation of new contracts, perform contract functionalities, and send ether to contracts or other participants. All of the transactions were indeed recorded on the blockchain, which is a public, allows to append the state of transaction on a data structure. The order in which transactions are recorded on the blockchain establishes the state of every contract also each user's balance. The smart contract's security has received significant interest among security agencies and researchers because it normally manages a lot of cryptocurrency worth billions of dollars.

Several strategies have been put forth to safeguard deployed contracts. With the help of examples like Sereum, which detects whether transaction executions are consistent with known vulnerability patterns and dynamically avoids reentrancy attacks by watching them. Sereum, however, can only deal with re-entrancy problems.

A. Security Flaws:

No doubt that Smart contracts on Ethereum network are popular but there are a few security concerns. Several flaws were uncovered by researchers, putting thousands of dollars at stake. This is alarming news, but there is a bright aspect to it as well. That's the reason still Ethereum is ruling the market.

Smart contracts cover a wide range of topics. Although templates can be utilized, any extra code poses a security risk. Since there are lot of Ethereum-based contracts out there, it's important to determine if they're safe. Fig. 1 shows how an attacker tries to attack smart contracts. Researchers hope to raise security standards by using a new approach to sniff out flaws. Unfortunately, already uncovered over 3,000 contracts that are currently susceptible. At current

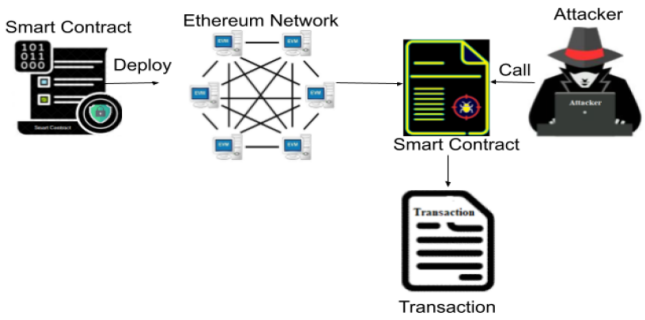

Fig. 1. How Attacker uses Ethereum Network

Ether pricing, these are worth approximately $6 million. Things might swiftly spiral out of control if someone took advantage of these weaknesses. The key issue is how smart contracts are utilized to manage the currency of other individuals. Even though it may appear to be interesting, there is always a cost. In reality existing contracts cannot be altered, this is a serious problem. There is little that can be done, especially if a security vulnerability is uncovered. For whatever reason, this is a severe design problem.

.

B. Motivation

An important turning point in the history of blockchain development was the use of smart contracts. Since Ethereum is a well-known platform Since then, blockchain can accommodate a variety of business requirements across a wide range of industries. As a result, smart contracts have been connected to an increasing number of digital assets. However, numerous attacks have resulted in significant losses for them. The sluggish mist community claims that 531,300,756.56 USD have been lost simply by hacking ethereum DApps. Even bad is that trust in blockchain has been eroding over time [4].

The Ethereum network's natural flow is frequently obstructed or even destroyed by attacks, which puts innovative technology at risk. The King of the Ether Throne assault and multiplayer games are further instances of Smart Contracts based attacks that come from flaws in already implemented Smart Contracts. As the market for Smart Contracts expands, attackers are drawn to it, necessitating the need to defend the company from such attacks [5]. Given that smart contracts are a relatively new technology, there are a number of vulnerabilities that are yet to be discovered. Hence a need to discover more vulnerabilities. With the existing Vulnerability detection tool like Oyente [6], there is a need to develop a tool that provides both detection of vulnerability and generating patches of the code to overcome them.

C. Our Contribution

We implemented a streamlined mechanism for vulnerability identification and smart contract deployment that can provide patches to close security gaps in already-deployed vulnerable smart contracts. We must fix contracts without changing their codes because nothing stored in blockchain can be tampered with, which is incredibly difficult and hard. The main concept of this study is to use independent smart contracts that have security rules (patches) incorporated in them to prevent fradulent transactions from happening in the first place, specifically to improve Ethereum Virtual Machine (EVM) to assist transactions verification and contracts binding. When presented with a weak smart contract, the streamlined interface for vulnerability identification and smart contract deployment first automatically creates a patch contract with secure rules based on the repair template, then deploys the patch on blockchain as a regular contract. Second, in order to repair the contract, the owner of the vulnerable contract must issue a unique transaction through our improved EVM. Finally, the fix that has been submitted will be used to verify any transaction that uses the vulnerable contract. The final visioned version of the mechanism can be seen more clearly in Fig.2. Afterward, transactions that may expose vulnerabilities will be stopped. Therefore, it is possible to defend against attacks on the original susceptible contract. Hence, the need to identify other vulnerabilities [7].Our method suggests both detection and code patching in conjunction with existing vulnerability detection tools like Oyente with tests for weaknesses in the written code, Smart Deploy is a blockchain-based deployment method for smart contracts that provides a consistent deployment interface for both layer 1 and layer 2 solutions [8].

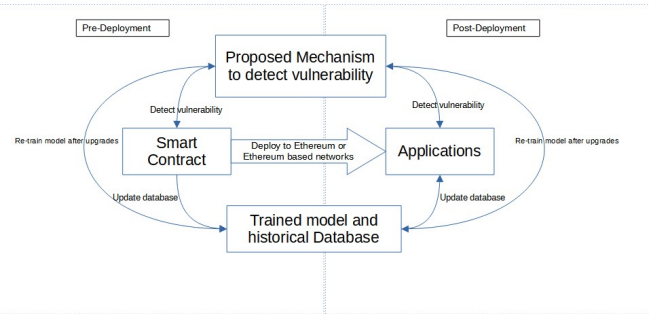

Fig. 2. Intended Final Vision of Mechanism

II. Methodology

The architecture of the proposed system, as seen in Fig.3, consists of two phases, a training phase and a detection phase. In the training phase, a sample of smart contracts are taken in order to train the model. They are first converted to Directed Acyclic Graph(DAG) after which the secondary and fallback nodes are removed to obtain the temporal and cluster nodes in the form of a matrix. This is then used along with a heuristic update formula to look for target key words that cause vulnerabilities in smart contracts which then updates the training model. This

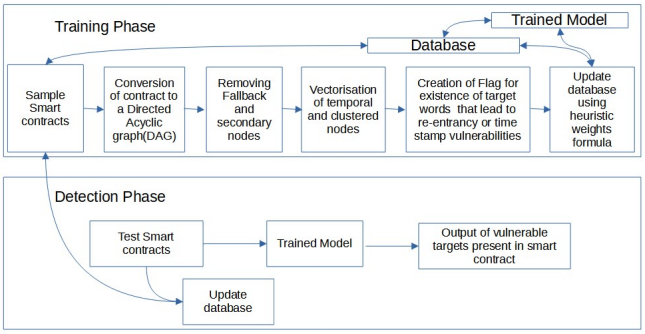

Fig. 3. Architecture of the proposed system

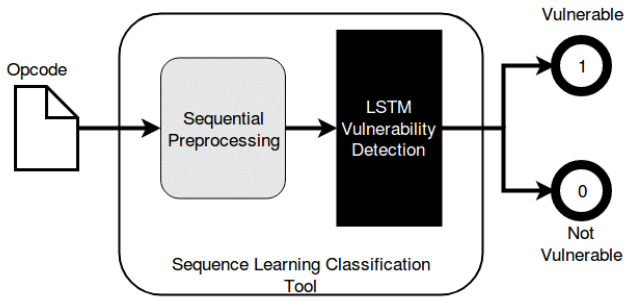

Fig. 4. Existing system to detect vulnerability

training occurs for over 100 epochs to give an initial base model.

The detection phase uses this trained model to look for vulnerabilities in the new smart contracts passed to the model and gives an output of the target vulnerabilities. The new smart contract is also stored in the smart contract database and the model is re-trained to achieve better accuracy and detection of wider range of vulnerabilities. The model in the proposed system is limited to re-entrancy and time stamp dependence vulnerabilities.

The proposed system is aimed at solving the absence of a singular interface/platform where all the features of multi-layer deployment, simple IDE for writing solidity code and vulnerability detection. The system-architecture of the proposed system is as follows :

1) Abstract Specification of sub-system:

- **Command Line Interface** : A simple interface for users to write and see code. It is through this interface that users can connect their wallets, deployment to different layers and test their code.

- **Database** : This contains the current historical data of different vulnerabilities and their respective patches of code. With new contracts being deployed, the system look for patterns and keeps testing them in a local test-net environment. New found vulnerabilities are then updated and fed back into the system.

- **Side-chain Node** : This node will operate off the mainnet, while acting as a gateway for users to interact with either layer 1 or layer 2.

- **Contracts** : Smart contracts are pieces of code that are used to interact with the blockchain. It is this code that has a lot of vulnerabilities and is susceptible to attacks.

In Figure 4, the current system for detecting vulnerabilities in smart contracts is depicted. The diagram makes it evident that various difference detection techniques are employed before a smart contract is deployed on the chain. The ultimate objective of the proposed mechanism is to pinpoint vulnerabilities after the smart contract has been deployed on the chain. [9].

1)Info Extractor Module: Which is responsible for extracting the Variable Dependence and Path Constraints.

2) Path Synthesis Module: Given all such data it is the responsibility of the path synthesis module to generate the patch template for the given Vulnerability.

3)Enhanced EVM: Miner Binds the Malicious Contract with Patch Contract, Thus preventing any instructions from accessing the malicious Smart Contract.

Figure 5 Shows three modules which are essential part of

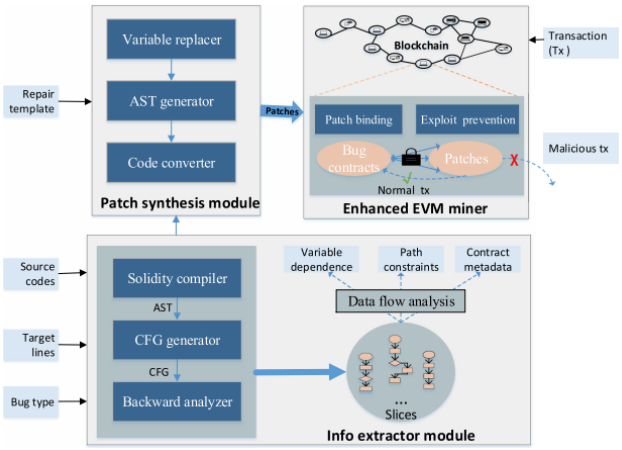

Fig. 5. Structure of Our Researched Technique

the technique proposed. We conducted extensive tests on actual smart contract functions, and the results demonstrate that our approaches consistently perform on par with existing tools for identifying various types of vulnerabilities that exist in smart contracts, such as re-entrancy and timestamp dependence vulnerabilities [9].

III. Trial of Implementing Vulnerability Detection from Proposed Method

A. Overview of Technologies used

1) Code Analyzer: A Solidity code analyzer is a tool specifically designed to examine Solidity code, the programming language used for Ethereum smart contracts. It

performs static analysis on the code to identify potential security vulnerabilities, code quality issues, and other potential pitfalls. By scrutinizing the code before deployment, a Solidity code analyzer helps developers improve the security, reliability, and overall quality of their smart contracts by providing actionable insights and suggestions for remediation, ultimately reducing the risk of vulnerabilities and enhancing the robustness of the contracts. To aid our research and projects, [10] Slither Static code analyzer has been used as reference.

2) Graph Neural Nets for creating a usable structure from smart contract's syntax: Graph Neural Networks (GNNs) are a class of neural network models designed to process and analyze graph-structured data. Unlike traditional neural networks, which operate on grid-like data structures like images or sequences, GNNs are capable of learning and reasoning about relationships and interactions between entities represented as nodes in a graph. GNNs iteratively update node representations by aggregating information from neighboring nodes, enabling them to capture and propagate information across the graph structure. This makes GNNs particularly well-suited for various tasks involving graph data, such as node classification, link prediction, graph classification, and recommendation systems, allowing them to effectively model complex relational dependencies and extract meaningful representations from graph-structured data.

3) 2D Convolution Neural Nets for training the model: 2D Convolution Neural Networks (CNNs) are a type of neural network architecture specifically designed to process two-dimensional grid-like data, such as images. By employing convolutional operations, pooling layers, and hierarchical feature extraction, 2D CNNs are capable of automatically learning and capturing spatial patterns and features from images. These networks excel at tasks such as image classification, object detection, and image segmentation, as they can effectively exploit the local connectivity and translation invariance properties inherent in visual data. With their ability to extract and analyze complex visual features, 2D CNNs have become a powerful tool for various computer vision applications.

B. Implementation details of modules

1) Code Analyzer - Parsing and feature extraction: To analyze Solidity code, which is the programming language used for developing smart contracts on the Ethereum blockchain before deployment, we make use of a static code analyzer to detect potential issues, vulnerabilities, and bugs in the code before it is deployed, helping developers improve code quality and security.

Here's an overview of how a static Solidity code analyzer typically works:

- Parsing: The analyzer parses the Solidity code to create an internal representation of the code structure. This step involves tokenizing the code, identifying variables, functions, modifiers, and other language constructs.
- Symbolic analysis: The analyzer performs a symbolic analysis to build a model of the code's behavior. It tracks the flow of values through variables, identifies dependencies between functions and contracts, and resolves references to external libraries and contracts.
- Rule-based analysis: The analyzer applies a set of predefined rules or patterns to identify potential issues and vulnerabilities in the code. These rules cover various aspects such as security vulnerabilities (e.g., reentrancy, integer overflow), coding best practices (e.g., code style, function visibility), and potential performance optimizations.
- Code quality assessment: The analyzer evaluates the code against established coding standards or style guides such as the Solidity Style Guide or the Ethereum Smart Contract Best Practices. It checks for adherence to naming conventions, indentation, documentation, and other coding style guidelines.
- Issue reporting: The analyzer generates a report detailing the identified issues, vulnerabilities, and code quality violations. It provides information about the location of the problematic code, a description of the issue, and in some cases, suggestions for remediation.
- Integration with development environments: Many static Solidity code analyzers integrate with popular development environments and tools, such as IDEs (Integrated Development Environments) or CI/CD (Continuous Integration/Continuous Deployment) pipelines. This allows developers to receive immediate feedback and easily incorporate the analyzer into their existing development workflows.

2) Training the model for feature extraction: To extract the feature, a training model comprising of Graph Neural Nets and 2D Convolutional Neural Nets are used.

- Graph Neural Nets(GNN)
  The goal here is to obtain a structure of the smart contract where all the secondary and fall back nodes are removed.The method of training Graph Neural Nets is as follows:

1) Data Preparation: The input data for GNN training consists of graph structures, where nodes represent entities, and edges represent relationships between nodes. The data may also include node features and edge attributes. The graph is typically represented as an adjacency matrix or an edge list.
2) Model Architecture: The GNN model architecture consists of multiple layers of graph convolutional operations. Each layer aggregates information from neighboring nodes and updates node representations based on the graph structure and node features. Common GNN variants include Graph Convolutional Networks (GCN), GraphSAGE, and Gated Graph Neural Networks (GGNN).

3) Forward Propagation: During training, the GNN processes the input graph through multiple layers by iteratively updating node representations. In each layer, the GNN performs message passing between neighboring nodes, incorporating information from the surrounding nodes and edges.
4) Loss Computation: Once the graph has been processed through the GNN layers, the final node representations are passed through one or more fully connected layers, and a loss function is computed. The choice of loss function depends on the specific task, such as node classification, link prediction, or graph classification.
5) Backpropagation and Parameter Update: The loss is backpropagated through the GNN layers and the fully connected layers to calculate the gradients with respect to the model parameters. The optimizer, such as stochastic gradient descent (SGD) or Adam, adjusts the parameters using the computed gradients, aiming to minimize the loss.
6) Iterative Training: The training process typically involves multiple iterations or epochs. In each epoch, the GNN is fed with different subgraphs or mini-batches from the training dataset to update the model parameters. This iterative process helps the GNN learn meaningful representations of nodes in the graph.

The flow of the above steps are shown in figure 6.

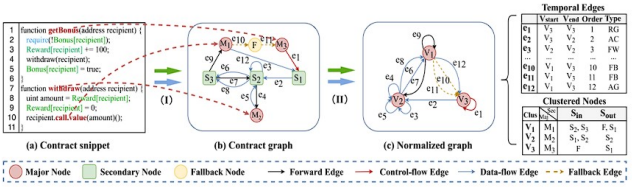

Fig. 6. Overview of obtaining trainable vectors

- 2D Convolutional Neural Nets(CNN) training :
  The input from GNN is taken, namely the removed secondary and fall back nodes, and the parsed data set is passed to a 2D CNN to train the model in the syntax of smart contract vulnerabilities. The method is as follows :

1) Data Preparation: The input data for 2D CNN training consists of two-dimensional grid-like structures, such as images. The data is typically represented as matrices of pixel values, where each pixel corresponds to a specific feature.
2) Model Architecture: The CNN model architecture consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers perform local receptive field convolutions to capture spatial patterns, while pooling layers downsample the feature maps to reduce dimensionality.
3) Initialization: The model's parameters (weights and biases) are initialized randomly or with pre-trained weights from another model if transfer learning is employed.
4) Forward Propagation: During training, the input images are fed into the CNN, and the feature maps are computed by applying convolutional operations, activation functions, and pooling operations. The feature maps capture hierarchical representations of the input images.
5) Loss Computation: Once the feature maps have been computed, they are flattened and passed through one or more fully connected layers, followed by an activation function. The output of the final layer is compared to the ground truth labels to compute the loss using an appropriate loss function, such as cross-entropy for classification tasks.
6) Backpropagation and Parameter Update: The loss is backpropagated through the CNN layers, and the gradients with respect to the model parameters are computed. The optimizer, such as SGD or Adam, adjusts the parameters using the gradients to minimize the loss.
7) Parameter Update: The model's parameters are updated using an optimization algorithm, typically stochastic gradient descent (SGD) or one of its variants. The gradients are used to adjust the parameters in the direction that minimizes the loss.The update is governed by the formula :

$$h_i^{t+1} = f\left(h_i^t W + \sum_{j \in N(i)} \frac{1}{c_{ij}} h_j^t U\right)$$

8) Iterative Training: Steps 4 to 7 are repeated for multiple iterations or epochs. In each epoch, the entire training dataset is passed through the network, and the parameters are updated accordingly. This iterative process allows the network to gradually improve its performance by adjusting the weights to minimize the loss.
9) Validation and Evaluation: Periodically, the model's performance is evaluated on a separate validation set to monitor its generalization ability and prevent overfitting. Metrics such as accuracy, precision, recall, or mean average precision are computed to assess the model's performance.
10) Hyperparameter Tuning: Various hyperparameters, such as learning rate, batch size, or regularization strength, may be fine-tuned to optimize the model's performance. This process often involves experimentation and validation set performance analysis.
11) Model Deployment: Once the desired level of performance is achieved, the trained model can be deployed to make predictions on new, unseen data.

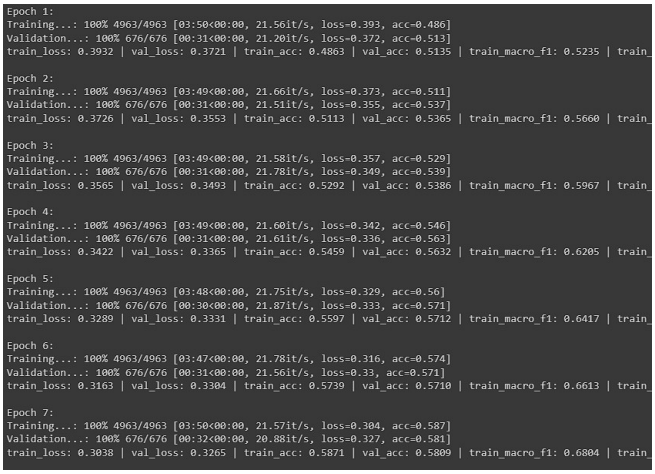

Fig. 7. training Epoch

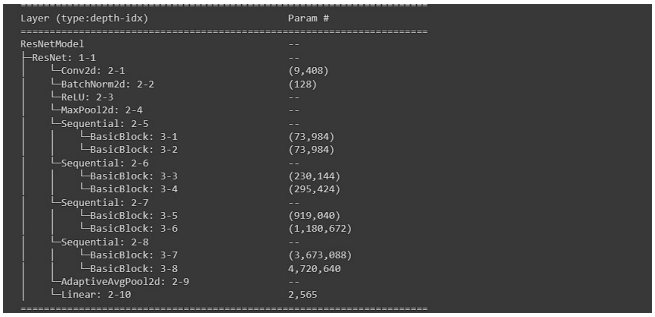

Fig. 8. Hyper-parameters

- Static code analysis
  After the features are extracted and the model is trained, the weights are trained model are passed as inputs for an already existing static analysis tool called Slither [11]. Slither is an open-source static analyzer for smart contracts. Using our new different trained model, we can make use of the existing robust tool to detect the vulnerabilities present in the passed smart contract.

C. Difficulties encountered and Strategies used to tackle

Smart contracts are are a very complex and versatile pieces of code that are like jigsaw puzzles when trying to evaluate the value/state of a particular vulnerable function. The main obstacles and points of difficulty that were faced during the project were due to the nature of smart contracts , and tailoring the training of machine learning models to smart contracts. This was due to a number of issues :

1) Lack of Complete Information: Smart contracts often interact with external contracts, libraries, and user inputs. Analyzing vulnerabilities requires knowledge about the behavior of these external components. However, obtaining complete information about the functionality and security of all dependencies can be difficult, especially when dealing with third-party contracts or closed-source libraries.
2) Complex and Evolving Attack Vectors: Smart contract vulnerabilities can stem from intricate attack vectors specific to blockchain environments. New attack techniques and vulnerabilities continuously emerge, requiring constant research and updates to vulnerability detection tools. Staying up to date with the evolving threat landscape is essential.
3) Solidity Language Complexity: Solidity, the language used for Ethereum smart contracts, has its own intricacies and features that may lead to subtle vulnerabilities. Ensuring comprehensive coverage of Solidity-specific constructs, edge cases, and vulnerabilities is a non-trivial task.
4) Lack of Standardized Security Frameworks: Although there are best practices and security guidelines available, there is no universally accepted standard for smart contract security. Different auditors, analyzers, or security tools may have varying approaches or focus areas, making it challenging to achieve consensus on vulnerability detection and mitigation.
5) False Positives and False Negatives: Automated vulnerability detection tools can produce false positives (identifying issues that are not actual vulnerabilities) or false negatives (missing real vulnerabilities). Striking a balance between minimizing false positives and ensuring accurate detection is crucial to avoid wasting developers' time on non-issues or overlooking actual vulnerabilities.
6) Context Sensitivity: Identifying vulnerabilities often requires considering the broader context of the contract, including its intended functionality, expected usage patterns, and potential interactions with other contracts. Analyzing this context accurately can be challenging, especially when dealing with complex or interconnected systems.

The above issues were overcome with the extensive help of work done in [1], [10], [12], [13], [14], [15], [16], [17], [18], [19], [20] and [21] without which the sought after integration would not have been possible. The initial target was set out to first deal with known vulnerabilities, like re-entrancy and timestamp attacks, that had a lot of resources available and sample smart contracts that could be used to train, test and validate our models.

IV. Experimental Analysis and Results

1) Evaluation Metric and Performance Analysis: In order to measure the effectiveness, efficiency and accuracy of our models in detection of vulnerabilities in smart contracts, we make use of metrics such as precision and recall, false positive and false negative rates, which helps to evaluate the reliability of the detection system, and benchmarking, which aids in comparing different detection techniques and identifying the most effective approaches. These evaluation methods enable robust analysis and improvement of vulnerability detection in smart contracts.

1) Precision and Recall: Precision measures the proportion of correctly identified vulnerabilities among all vulnerabilities detected. Recall, on the other hand, measures the proportion of correctly identified vulnerabilities among all actual vulnerabilities present. These metrics help assess the accuracy and completeness of the vulnerability detection.
2) False Positive and False Negative Rates: False positives occur when a vulnerability is incorrectly identified, while false negatives occur when a vulnerability is missed. Evaluating the false positive and false negative rates helps determine the level of accuracy and reliability of the vulnerability detection system.
3) Benchmarking: Benchmarking involves comparing the performance of different vulnerability detection tools or techniques using standardized datasets and evaluation metrics. It helps identify the most effective approaches and highlights areas for improvement.

2) Experimental Data set: To aid the research and work of the project, open source data sets are used to generate the vulnerability database, a set of contracts that have known vulnerabilities in them. One such sample contract, in which a known vulnerability is present, namely re-entrancy. Take an example smart contract with known vulnerability where the **migration.restricted()** function , on lines 11 - 13 in the code below, does not always execute. :

```
pragma solidity >=0.4.21 <0.7.0;

contract Migrations {
  address public owner;
  uint public last_completed_migration;

  constructor() public {
    owner = msg.sender;
  }

  modifier restricted() {
    if (msg.sender == owner) _;
  }

  function setCompleted(uint completed)
  public restricted {
    last_completed_migration = completed;
  }
}
```

```
INFO:Detectors:
Modifier Migrations.restricted() (Migrations.sol#11-13) does not always execute
 _; or revertReference: https://github.com/crytic/slither/wiki/Detector-Documen
tation#incorrect-modifier
```

Fig. 9. Migration error

In the above given code, another vulnerability is using of older versions of solidity which do not have upgraded protocols to prevent attacks on smart contracts.
Similarly, the model also detects arithmetic vulnerabilities present in a smart contract. Take the below code snippet for

```
INFO:Detectors:
Pragma version>=0.4.21<0.7.0 (Migrations.sol#1) allows old versions
Pragma version^0.5.16 (MultiSigWallet.sol#1) allows old versions
solc-0.5.16 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorr
ect-versions-of-solidity
```

Fig. 10. Vulnerability due to use of older version of solidity

example. Here, the vulnerability is the array length has been predefined or is a user-controlled value. :

```
    contract MultiSigWallet {
    event Deposit(address indexed sender,
    uint amount, uint balance);
    event SubmitTransaction(
        address indexed owner,
        uint indexed txIndex,
        address indexed to,
        uint value,
        bytes data
    );
```

```
INFO:Detectors:
MultiSigWallet (MultiSigWallet.sol#3-237) contract sets array length with a use
r-controlled value:
        - owners.push(owner) (MultiSigWallet.sol#75)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#array-
length-assignment
```

Fig. 11. Array Length vulnerability

The detection of this vulnerability relies on our model's training to identify specific markers or values within the syntax of the written code, like **uint256**, or **uint ¡array variable¿**. A combination of these markers are known to cause vulnerabilities, and using this as reference the smart contract is search and compared with these markers, returning matches when found.

The most imminent vulnerability in smart contracts is re-entrancy, which has caused loss of massive funds from users. The given code below has a vulnerability that allows an external user or smart contract to re-enter a deployed contract, calling its functions causing loss of funds.

```
function executeTransaction(uint _txIndex)
public onlyOwnertxExists(_txIndex)
notExecuted(_txIndex){
    Transaction storage transaction =
    transactions[_txIndex];
     require(transaction.numConfirmations >=
     numConfirmationsRequired,
    "cannot execute tx"
    );

    transaction.executed = true;

    (bool success, ) =
    transaction.to.call.value(transaction.value
    (transaction.data);
    require(success, "tx failed");
```

The sample output from the attempted system is shown in the figure below.

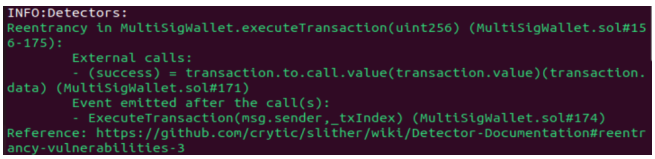

Fig. 12. Re-entrancy Vulnerability

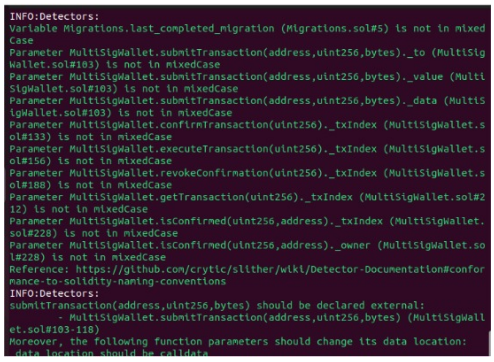

Fig. 13. Output of attempted methodology

V. Conclusion and Future Enhancements

A. Conclusion

The emergence of blockchain technology has revolutionized trustless communication, reshaping traditional business models by eliminating the reliance on intermediaries. However, Ethereum blockchain-based smart contracts carry inherent vulnerabilities that pose substantial risks, potentially resulting in severe financial losses. The persistent challenges in securing smart contracts have impeded the widespread practical application of blockchain. While current research predominantly concentrates on detecting vulnerabilities off-chain, the irreversible nature of on-chain smart contracts demands innovative solutions. In response, we introduce the "Streamlined Interface for Detection of Vulnerabilities and Deployment of Smart Contract." This groundbreaking approach provides an automated mechanism to patch deployed contracts vulnerable to exploitation without modifying their original code. Through the generation of patch contracts to proactively prevent malicious transactions, our streamlined interface aims to bolster the security of on-chain smart contracts. Our emphasis on critical bug types, including reentrancy, arithmetic bugs, and unchecked low-level checks, underscores the efficacy of our proposed interface in autonomously rectifying vulnerabilities within deployed smart contracts. This innovative solution seeks to reinforce the integrity of blockchain-based systems, creating a more secure and reliable environment for the practical implementation of smart contracts. As the blockchain landscape evolves, the Streamlined Interface for Detection of Vulnerabilities and Deployment of Smart Contract emerges as a promising advancement in ensuring the future security of decentralized and trustless transactions. From the work done, the smart contracts passed to the model had a total of 20 known vulnerabilities present, out of which the model has detected 19 vulnerabilities, achieving a **95%** accuracy from the given set.

B. Future Enhancements

Summary of future development :

- Integration of new chains, with their respective methodologies.
- Making decentralised server/nodes available to deploy and maintain applications.
- Working on developing a methodology that will work as a dynamic smart contract analyzer, that will check for vulnerabilites after deployment.

References

[1] Sudarshan Parthasarathy, Akash Harikrishnan, Gautam Narayanan, Lohith J. J, and Kunwar Singh. Secure distributed medical record storage using blockchain and emergency sharing using multi-party computation. In *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2021.

[2] Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4(2):15, 2008.

[3] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[4] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[5] Deepak Prashar et al. Analysis on blockchain vulnerabilities & attacks on wallet. In *2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, pages 1515–1521. IEEE, 2021.

[6] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[7] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. Blockscope: Detecting and investigating propagated vulnerabilities in forked blockchain projects. *arXiv preprint arXiv:2208.00205*, 2022.

[8] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*, pages 201–226. Springer, 2020.

[9] A Averin and O Averina. Review of blockchain technology vulnerabilities and blockchain-system attacks. In *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, pages 1–6. IEEE, 2019.

[10] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[11] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, May 2019.

[12] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[13] TonTon Hsien-De Huang. Hunting the ethereum smart contract: Color-inspired inspection of potential attacks. *arXiv preprint arXiv:1807.01868*, 2018.

[14] Martina Rossini, Mirco Zichichi, and Stefano Ferretti. On the use of deep neural networks for security vulnerabilities detection in smart contracts. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 74–79. IEEE, 2023.

[15] Chavhan Sujeet Yashavant, Saurabh Kumar, and Amey Karkare. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. *arXiv preprint arXiv:2202.11409*, 2022.

[16] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *International symposium on automated technology for verification and analysis*, pages 513–520. Springer, 2018.

[17] K Anusree Manoj P Guru Nanma Srinivasan Pooja J, Lohith J. Tp-detect: trigram-pixel based vulnerability detection for ethereum smart contracts. *Multimedia Tools and Applications*, 2023.

[18] Singh Kunwar Chakravarthi Bharatesh J J, Lohith. Digital forensic framework for smart contract vulnerabilities using ensemble models. *Multimedia Tools and Applications*, 2023.

[19] Singh Randeep Mir Bilal Ahmed J Lohith J Chakravarthi Dhruva Sreenivasa Alharbi Adel R Kumar Harish Hingaa Simon Karanja Kumar, Vijay. Smart healthcare system with light-weighted blockchain system and deep learning techniques. *Computational Intelligence and Neuroscience*, 2022.

[20] G. Kannan, Manjula Pattnaik, G. Karthikeyan, Balamurugan E, P. John Augustine, and Lohith J J. Managing the supply chain for the crops directed from agricultural fields using blockchains. In *2022 International Conference on Electronics and Renewable Systems (ICEARS)*, pages 908–913, 2022.

[21] Abhishek H Bharath Kumar Lohith J. J Nikhil C, Hetav Pabari. A review on supply chain management in agriculture empowered by ethereum and ipfs. *International Conference on Computational Intelligence and Digital Technologies*, 2021.