ISSN: 2633-4828 Vol. 4, No.2, September, 2022

International Journal of Applied Engineering & Technology

A Machine Learning-Based Dynamic Method for Detecting Vulnerabilities in Smart Contracts

Jasvant Mandloi[1], Pratosh Bansal[2]

Department of Information Technology, Institute of Engineering and Technology, Devi Ahilya Vishwavidyalaya, Indore, India.

jasvant28284@gmail.com, pratosh@hotmail.com

Date of Submission: 21st July 2022 Revised: 14th September 2022 Accepted: 06th Oct 2022

How to Cite: ***Jasvant Mandloi et.al.,(2022). A Machine Learning-Based Dynamic Method for Detecting Vulnerabilities in Smart Contracts. International Journal of Applied Engineering &Technology 4(2), pp.110-118.***

Abstract **- Real-world application development through Smart Contracts on the Ethereum Blockchain platform is one of the emerging technologies. It also has much vulnerability, and reentrancy is among the most popular ones. In our work, we have reviewed the tools based on ML for vulnerability detection in Ethereum smart contracts. Based on that, we proposed a framework that can dynamically monitor threats based on the blockchain platform's transaction meta-data and balance data. It does not require any changes or updates to the existing system and does not require expertise to implement. This framework will extract features for machine learning classifier models from the transaction data and identify the transaction as agreeable or unfavorable. It will help to identify the reentrancy threat as well as the cause of it and help the developer to trace it from where the attack is generated. In the ML classifier for the framework, random forest and decision tree are used. The cumulative performance of both is 98 percent on 540 transactions.**

Index Terms **- Smart Contract, Reentrancy, Vulnerability, Attacks, Transaction, Ethereum**

INTRODUCTION

A blockchain is a public ledger that manages user assets. A smart contract contains the rules for transferring digital assets stored in the ledger. The exchanges happen inside blockchain transactions that are constantly stored. As a result, smart contracts may be used in various applications, including financial and governance [1]. Currently, in May 2022, Ethereum has a valuation of over $130 billion, and it is in the second position after Bitcoin in the world [2]. Smart contracts on Ethereum allow users and others to communicate by invoking functions in their contracts. Ethereum smart contracts are usually written in Solidity, a language very much like JavaScript. In that execution, the cost is calculated with the unit known as gas, which is required to carry out the activities on a smart contract. Gas costs are denominated in the Ether unit, Ethereum's native currency [3].

Challenges to ensuring the proper behavior of the contract arising from a new technique to construct smart contracts. It makes the code susceptible to security breaches that can be used by other account holders in the Ethereum network.

Recently, many attacks have been performed on the Ethereum network, causing millions of Ether to be lost. The most famous attack is DAO (Decentralized Autonomous Organization), performed using the reentrancy vulnerability. The outcome of this incident will result in the loss of 3.5 million ETH, equivalent to 50 million USD [4].

The reentrancy attack involves repeatedly calling the same function or group of functions before the calling is finished. In such an integrated invocation scenario, smart contracts may behave in a different way than the expected one. It helps the attacker use this flaw to transfer the fund's ETH from the victim's account to another. Reentrancy is one of the most severe vulnerability categories in the Ethereum smart contracts. In the present scenario, approaches for detecting reentrancy flaws are based on investigating the financial asset exchanges and control flows in smart contracts using extensive code analysis with bespoke criteria. However, in the real world, there are very few opportunities to exploit such attacks at the transaction level. We have tried to perform it in different ways.

Extraction of Ethereum's Raw Data

Fig. 1 depicts the typical Ethereum transaction execution process from Block N to the EVM through a blockchain peer. We may acquire three forms of blockchain raw data: block, receipt, and trace. In our work through Trace, features for the ML model can be derived [5].

Our framework is designed to analyze the transactions on the smart contract and, based on them, analyze the malicious transactions. A framework for identifying reentrancy flaws in Ethereum smart contracts. The blockchain system's transaction meta-data and account data would be the only data sources for this architecture. It employs a machine learning model to identify transactions as benign or hazardous based on attributes extracted from transaction data. In the framework, the random forest and decision tree models are used. They have 97% accuracy for the 100 transactions. Our work is organized in the paper as follows: Section II details smart contracts and reentrancy attacks. After that, Section III will cover related work to our work. Section IV outlines our framework process and approach to executing it.

Our experimental setup and results with comparison are in sections V and VI. In in the last section, VII is concluded with future work.

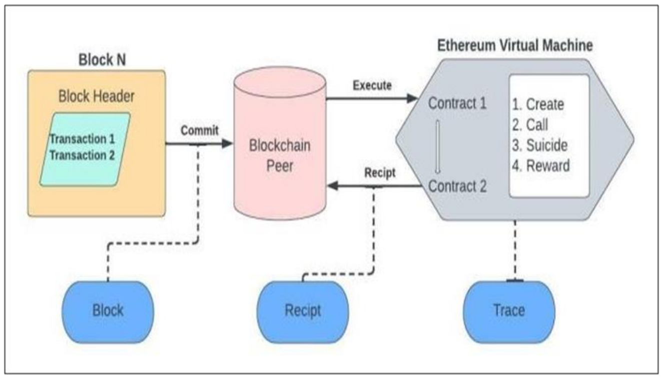
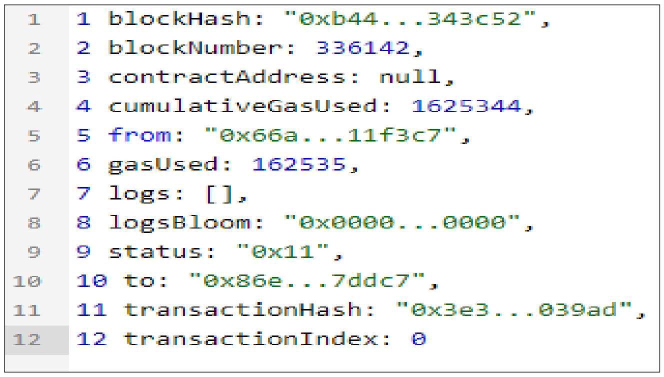


Figure 1: Ethereum Row Data Collection Scenario

In the first step at the Ethereum blockchain level, we monitor transactions in real-time. Figure 3 displays an example transactional trace from which we collected information for the classification model. Such an inspection



```
blockHash: "0xb44...343c52",
blockNumber: 336142,
contractAddress: null,
cumulativeGasUsed: 1625344,
from: "0x66a...11f3c7",
gasUsed: 162535,
logs: [],
logsBloom: "0x0000...0000",
status: "0x11",
to: "0x86e...7ddc7",
transactionHash: "0x3e3...039ad",
transactionIndex: 0
```

eliminates the need for a thorough analysis of the smart contracts themselves and enables us to apply our idea immediately on the Block chained client-side without affecting the smart contracts or the client-side.

After that, on the monitored transaction metadata, we apply machine learning techniques. It eliminates the requirement to create (potentially faulty) rules and paves the road for future vulnerability detection and correction.

Background

Ethereum Smart Contract

Figure 2: An illustration of a transaction receipt taken from the Ethereum blockchain

Ethereum-based smart contracts are autonomous, decentralized blockchain-based programs that specify the contract terms between market participants, doing away with the need for third parties to act as trusted brokers and arbitrators.

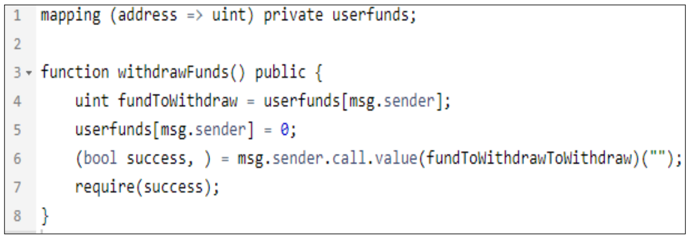
These compact applications developed in the Ethereum bytecode are represented in a particular format [6]. The bytecode is the outcome of the compiled code in a language such as, I.e.; The code is executed step by step on the virtual machine, one instruction at a time. The reason is that each instruction has a value that can be measured in the form of gas; the caller of the smart contract has to pay for it. The virtual machine's responsibility is to manage instruction outcomes and their impact in the form of cost on each asset on the blockchain network. There is always the possibility of errors in the entire process, from sudo code to byte code. Potential vulnerabilities may occur, and reentry is one of the possible attacks that could occur [7].

Reentrancy Vulnerability Attack,

One of the characteristics of smart contracts is their capacity to call external contract codes, and they have the power to deliver digital money for transactions to external user addresses. Calls like these to outside contracts might lead to reentrancy. In Ethereum, ether can be exchanged between two parties. When a contract receives a message including ether but has no data, and no function is specified, an anonymous default function, the fall-back function, is called. In this case, a contract invited by calling another account can specify how much gas the called party is permitted to spend. If the target account is in the form of a contract, it will be executed and will be able to use the gas budget that has been allotted. In such a scenario, if the contract is vicious and the gas budget is high enough, the caller can be called back as a reentrant call. If, for instance, the caller's functionality is not re-entrant due to failing to update the internal state holding balance information, the attacker can use this vulnerability to siphon money from the susceptible contract. [8].

Reentrancy attack on a single function

The oldest known instance of this problem had procedures that may have been invoked more than once before the first invocation was finished. This could lead to harmful interactions between the function's different invocations [9].

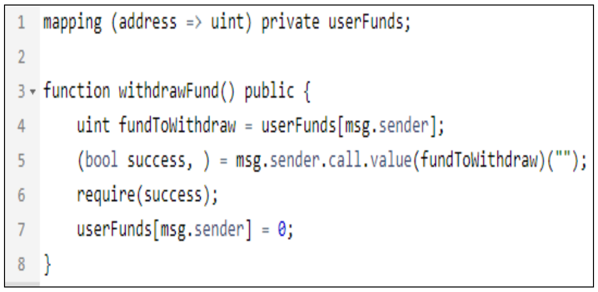


```
mapping (address => uint) private userfunds;

function withdrawFunds() public {
    uint fundToWithdraw = userfunds[msg.sender];
    userfunds[msg.sender] = 0;
    (bool success, ) = msg.sender.call.value(fundToWithdrawToWithdraw)("");
    require(success);
}
```

Figure 3 An illustration of a transaction how to avoid reentancy Attack in Ethereum blockchain

The user's fund is not set to 0 until the end of the function, so the second (and subsequent) invocations will continue to succeed and withdraw the amount. In the above case, the best way to avoid this attack is to ensure that you don't call an external function until you've completed all the necessary internal tasks.



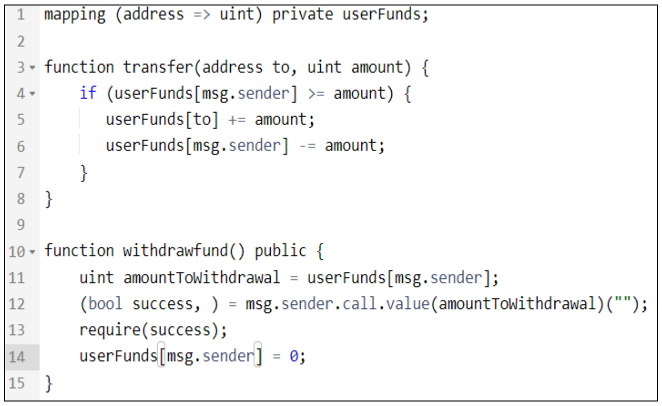
```
mapping (address => uint) private userFunds;

function withdrawFund() public {
    uint fundToWithdraw = userFunds[msg.sender];
    (bool success, ) = msg.sender.call.value(fundToWithdraw)("");
    require(success);
    userFunds[msg.sender] = 0;
}
```

Figure 4: Reentry Attack Avoidance

Cross-function Reentrancy

An attacker may also be able to carry out a similar assault by combining two functions into the same state.



```
mapping (address => uint) private userFunds;

function transfer(address to, uint amount) {
    if (userFunds[msg.sender] >= amount) {
        userFunds[to] += amount;
        userFunds[msg.sender] -= amount;
    }
}

function withdrawfund() public {
    uint amountToWithdrawal = userFunds[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdrawal)("");
    require(success);
    userFunds[msg.sender] = 0;
}
```

Figure 5: Cross-site reentrancy attack

When the attacker's code is performed on the external call to withdraw funds, they call transfer (). They can transfer the tokens even though they have already received the withdrawal since their balance value has not yet been reset to zero. In addition, the DAO attack made use of this issue.

One way to avoid reentrancy is through solidity function modifiers, which might be used to make checks before handing control over to the fallback function of another contract. Before providing ether to the interacting contract, it examines and updates its status.

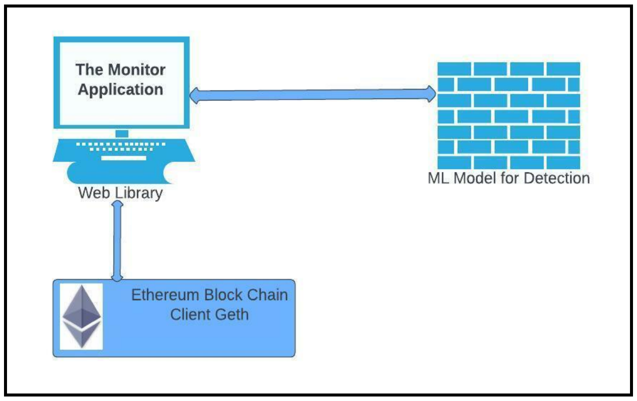


Figure 6 Framework to detect the reentrancy attack

RELATED WORK

For vulnerability analysis, there are primarily four different types of methodologies: static and dynamic code, systematic specification and validation, and other chunks also [10].

At the program or code analysis level to detect the potential vulnerability, detection can be performed mainly through static and dynamic analysis. In the first one, the source code is analysed without running it, and in the second one, the code is analyzed based on the program's behaviour in the running state.

The main advantage of the first one is that it can detect flaws in the code at the exact position, and it's done by skilled software assurance developers that know the code inside and out. Also, repairs may be completed more quickly, especially if automated techniques are employed. But it also has some major disadvantages: it is time-consuming if done manually, and automated tools are not there to support all programming languages. These automated tools produce false-positive and false-negative results. There are not enough trained personnel to thoroughly conduct static code analysis. Another disadvantage is that it does not find vulnerabilities introduced in the runtime environment. On the other hand, dynamic analysis works on the real-time system and detects flaws in the code during runtime. It sometimes might not be effective in locating the necessary inputs to bring about this. Dynamic analysis has a certain advantage like automated tools give flexibility in what to search for and find vulnerabilities in a runtime context. It also enables the study of programs for which you lack access to the source code, which works with any program.

In the present scenario, 75% of the tools are static, and 23 % are dynamic analysis tools. Only 2% of the tools are hybrid [11].

Table-I
The most commonly used static tools for smart contracts are [06], [11], [12], [13], and [14].

Tools	EVM Byte Code (as Input)	Solidity Code as Input	Support both	Platform	Analysis Approach
Echidna	N	N	N	Haskel	Fuzz testing
FEther	N	Y	N	Coq	Symbolic Execution
ESCORT	Y	N	Y	-	Machine learning
Ether(S-GRAM)	Y	N	Y	Python	Machine learning
GasTap	N	Y	N	Python	Symbolic Execution
GASOL	Y	Y	Y	-	Code Instrumentation
Pakala	N	Y	N	Python	Symbolic Execution
SAFEVM	Y	Y	Y	Python	Constraint Solving, Symbolic Execution
SIF	N	Y	N	C++	Code Instrumentation
Slither	N	Y	N	Python	Constraint Solving, Code Transformation
Smartbugs	N	Y	N	Python	Machine learning
Smartcheck	N	Y	N	Java	Code Transformation
SmartEmbed	N	Y	N	JavaScript	Code Transformation, Code Instrumentation
VeriSmart	N	Y	N	Ocaml	Code Instrumentation
Solidifier	N	Y	N	-	Formal Verification
E-EVM	Y	N	N	Python	Symbolic Execution
Erays	Y	N	N	Python	Code Transformation
Mythril	Y	N	N	Python	Constraint Solving, Symbolic Execution
Octopus	Y	N	N	Python	Symbolic Execution, Fuzz Testing
Osiris	Y	N	N	Python	Symbolic Execution
Oyenet	Y	N	N	Python	Symbolic Execution
Securify	Y	N	N	Java	Abstract interpretation
Vandal	Y	N	N	Python	Symbolic Execution

Table-II
The most commonly used dynamic tools for smart contracts are [06], [11], [12], [13], and [14].

Tools	EVM Byte Code (as Input)	Solidity Code as Input	Support both	Platform	Analysis Approach
ContractLarva	N	Y	N	Haskell and Tex	Code Instrumentation
Ethlint	N	Y	N	Java Script	Code Instrumentation
Harvey	N	Y	N	-	Fuzz testing
Modcon	N	Y	N	JavaScript	Model-Based Testing
Solitor	N	Y	N	Java	Code Instrumentation
ContractGuard	Y	N	N	Java Script	Machine Learning
EthBMC	Y	N	N	Rust	Symbolic Execution
Etherolic	Y	N	N	Rust	Fuzz testing, Taint analysis
EVMFuzz	Y	N	N	Python	Fuzz testing
Manticore	Y	N	N	Python	Symbolic Execution
EASY FLOW	Y	Y	Y	Go	Taint Analysis
ReGuard	Y	Y	Y	Python	Fuzz testing

In recent times, the public has been more concerned about the security of smart contracts, and some progress has been made by using machine learning techniques to find contract vulnerabilities. Some popular tools for using ML for reentrancy are very few available in the present scenario in static, dynamic, and hybrid, i.e., Contract Guard, Smart Bugs, ESCORT, and Ether(S-GRAM).

Some work was done earlier using machine learning for vulnerability detection in that the first one found was SMARTBUGS in the year 2020; it is developed in an open-source style on a publicly accessible platform, with the framework being built in Python. It supports a variety of tools for evaluating smart contracts developed on a blockchain platform. The command-line interpreter, tool configuration, tool docker images, dataset, and SMARTBUGS runner make up the first four components of the complete framework. For user interaction, SMARTBUGS supports both the web interface and the command-line interface [12]. It's a tool for dynamic analysis created using JavaScript in 2019. It implements the notion of an anomaly-based intrusion detection system. Its fundamental job is to alarm the smart contract administrators whenever they notice any suspicious behavior and roll the smart contract back to its prior secure state, ContractGuard[13]. Next to our work is ESCORT [06], a static analysis tool published in the year 2022. Deep Neural Network-based methods are used in the vulnerability detection framework for Ethereum blockchain smart contracts. It has expandable and simple features as it works on lightweight transfer learning on unobservable security problems. The ESCORT framework has two major components. The first component extracts the semantics and characteristics of the Ethereum-based smart contract, and followed by the second one, which consists of many branch structures, and accepts input given by the 1st component's features. This multiple branch structure will look at individual security threats. Another one is Ether (S-GRAM). It is a framework that has features like security with semantic awareness. It was developed based on the S-Gram artifact and was created in Python in 2018. It has two operation steps to detect vulnerabilities: build the model and then check the security. It blends the N-gram language model with compact static semantic labeling to acquire patterns in the data of contract tokens. So it captures high-level semantics to foretell future vulnerabilities [15]. One more framework, Slither, built on static analysis, provides detailed information on Ethereum smart contracts. Its basic operation involves converting Solidity smart contracts into an intermediary state known as Slith IR. To make analysis implementation easier while protecting semantic data that can be lost during the conversion of Solidity to byte code, Slith IR employs an (SSA) Static Single Assignment Form and a reduced instruction set. It was used to apply widely-used program analysis techniques, including data flow and taint tracking. The framework has four major use cases: automatic vulnerability discovery, automatic code optimization opportunity detection, enhancement of the user's knowledge of the contracts, and help with code review [16]. Another framework that will support the mutation of more than one transaction is by detecting basic exploitation via an oracle that keeps track of each smart contract instance's balance, doing away with the necessity for certain software patterns to find vulnerabilities[17].

There is a lot of scope for applying ML for vulnerability detection in smart contracts. Our work is the way to utilize ML in finding problematic execution patterns in smart contracts, with an emphasis on reentrancy.

METHODOLOGY

The dynamic framework detects and discovers reentrancy problems in deployed smart contracts without requiring their source code. This framework needs to consider dynamic behavior, and dynamic response is retrieved using metadata specifying how contracts interact. This monitoring is based on the present application programming interface of the unmodified Ethereum blockchain client (API).

Steps to Implement It

Step 1. Input: The dataset, Smart Contracts dataset, is implemented as input. The dataset is collected from a dataset repository.

Step 2. Data selection: In this step, we can select the input data using the panda package[18][19].

Step 3. Preprocessing: The collected input data is subjected to preprocessing.

In the preprocessing step,

- It can handle the missing data.
- It can perform label encoding.

Step 4: Data Splitting: The preprocessed data is split into training and testing sets for decision-making.

- Train data set is used to evaluate the model.
- Test data set is used to predict the model.

Step 5. Classification: In this step, we can implement the different machine learning classification algorithms, such as

- Random Forest (RF) Algorithm
- Decision Tree (DT) Algorithm

Step 6. Output: In this step, we can detect the vulnerability using classification algorithms.

The framework consists of two parts.

1. The scanner keeps track of blockchain transactions.
2. The pointer, which distinguishes between benign and malevolent behavior.

Table-III:
The Pointer ML model uses features collected by the scanner.

Features	**Monitoring Mechanism**
Gas_Usage of Transaction	Event Subscription
One fund distinction	Probing
Differences between the two funds	Probing
Avg Call_Stack_depth	Probing

Before the deployment of our technique to identify fraudulent production transactions, it is trained on a training set and a variety of classifiers against which the scanner can be customized.

Scanner:-

The scanner connects with the Ethereum blockchain client to collect data on specified transactions. It connects to and queries the Ethereum network using the most recent version of web3.py[18], the interface provided to interact with Ethereum.

The following data was gathered via the scanner:

Subscription to events that the Ethereum client sends out.

- When a transaction is returned to an account, these events are released. We use pending transactions in our job for any new transactions pertaining to the accounts we are keeping an eye on.

Probing the blockchain regularly until the desired data is obtained.

- It is appropriate to obtain details regarding a previously mined transaction or determine the contract status following an event.

Pointer:

The system's detector is the element that distinguishes between risky and safe transactions. A component of the machine learning model is taught as the monitor sends in data and analyzes and sanitizes the data it receives.

Feature Extracted

The table [3] lists the extracted features and how they were monitored.

The difference between a contract's funds before and following a transaction is what is meant by the feature known as the contract fund difference. The functionality contract fund differences may just be swapped out for any other asset the contracts are transferring to satisfy the needs of the specific use case.

The only feature directly extracted from the transaction trace is the average call stack depth. The value of this feature won't be significantly altered by calling a standard function inside a contract. External calls made iteratively will dramatically change this number, though. This is frequently true for the reentry vulnerability, in which the offender contract continues to run by repeatedly calling a specific function in the recipient.

It makes sense that this characteristic would be a strong indicator that the transaction is detrimental. An attacker can easily evade detection by restricting the number of recursions. Furthermore, to reduce modeling bias and make it harder for the detector to identify hazardous transactions, we agreed to attempt to arbitrarily reduce its average call stack depth for those transaction data. As was already stated, gas is used during the blockchain execution of contract code.

A contract's specific actions as part of a transaction determine how much gas is used. We utilize this gas usage to comprehensively describe the operation because a major attack on a susceptible contract may display a particular execution pattern.

Classifier

We trained and evaluated the following models in our scanner to determine which was the best.
To be added:

- Random Forest (RF)
- Decision Tree (DT)

Each of our models was created with the latest Pandas-Pythonata Analysis Library[19].

Framework application with a Smart Contract:

Consider creating an application and then putting it into use in the smart contract on Ethereum. The developers of smart contracts may protect their smart contracts using our framework. They set up this framework on their personal computer and communicate with the Ethereum network to monitor the status of their executed contract. The framework will gather and process the transactions' information as they are sent to the monitored smart contract. The machine learning model is then taught to classify transactions as beneficial or detrimental, which later can be used to give feedback to the programmer or as part of a system for managing vulnerable contracts and safety information. That can also alert administrators to users or stop the usage of a susceptible contract.

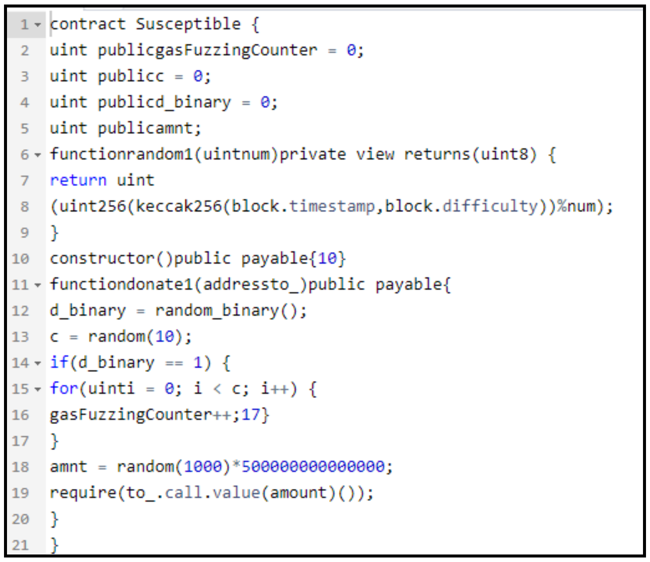
Experimental Setup

For our tests, we selected 25 open-source contracts that carry out a certain operation, which we refer to as "service contracts" in this document. These contracts were initially utilized in[20,21], and the source code is available on Etherscan. It employs 20 smart contracts to access and make use of the features. In those service contracts (13 robust contracts and 13 vulnerable contracts) and user contracts (11 friendly contracts and 9 suspicious contracts). A user contract may be benign or malevolent, and a service contract might be secure (unexploitable) or have a weakness. The service contract's vulnerability can only be discovered if a malevolent user and a weak service contract are combined. For the experiment, we kept track of 540 sequences of numerous transactions, 290 of them were safe, and 250 of them were hazardous. Before the experiment began, each of these transactions had a manual label applied, making them suitable for training/testing classification and prediction. At this stage, we give our classifier labeled transaction data that is stored offline; during production, unlabeled online data may be used. Among the 540 transactions, 140 were chosen from among 25 open-source service contracts, wherein 20 new user contract versions were added.

Two pairs of contract templates are utilized (four contracts), the remaining 400 transactions are produced randomly, including both harmful and benign transactions. To produce these arbitrary transactions, the service and the user enter into contracts that fuzz their conduct to represent a range of behaviors in actual occurrences. A sophisticated internal calculation may be involved with a specific call stack depth or gas use. This is another justification for having unpredictable behavior (fuzzing) in service and user contracts. It can make an attack more difficult to spot. We would like to include this behavior in our data to get a less skewed classifier in the detector. As a result, these transactions are created in a way that avoids the majority of the cases. As an illustration, we add a random loop with a 50% chance into the weak contract template to fudge the gas use, i.e., lines 12-18 Figure 7. Every time the counter is used, more gas is used. To make it more difficult, we additionally randomize the user's overall payment amount and the frequency of attacks employing reentrancy.

- Because every contact between a service and its users has the potential to be beneficial or detrimental. The following scenarios might happen.
- The user contract attempts to take advantage of a reentrancy vulnerability (that might or might not be present in the service contract) but fails. A few of the following outcomes will result from this.
    - The Ethereum run-time environment reverses the transaction's effects on the target contract state. The Ethereum monitoring API does not make public these failed (reverted) transactions and, as a result, are not considered in our study.
    - The transaction is not reversed; rather, it retains its intended initial impact: favourable exchange.

As was previously said, data that the scanner has gathered will be passed to the detector for categorization. With the help of the data mentioned above, we trained and validated the detector models. We employed stratified 10-fold cross-validated training and test sets for each model to obtain consistent and dependable findings. The entire experiment (such as the cross-validation) was run ten times for every number in the plots, and the average performance was calculated.



```
1  contract Susceptible {
2  uint publicgasFuzzingCounter = 0;
3  uint publicc = 0;
4  uint publicd_binary = 0;
5  uint publicamnt;
6  functionrandom1(uintnum)private view returns(uint8) {
7  return uint
8  (uint256(keccak256(block.timestamp,block.difficulty))%num);
9  }
10 constructor()public payable{10}
11 functiondonate1(addressto_)public payable{
12 d_binary = random_binary();
13 c = random(10);
14 if(d_binary == 1) {
15 for(uinti = 0; i < c; i++) {
16 gasFuzzingCounter++;17}
17 }
18 amnt = random(1000)*500000000000000;
19 require(to_.call.value(amount)());
20 }
21 }
```

Figure-7 Smart contract's source code, which was utilized to produce arbitrary damaging transactions for the test.

Result & Discussion

We develop and evaluate two distinct classifiers and contrast them based on accuracy, F1 score, recall, and average false positive and false negative rates (FPR and FNR).

In our work, the decision tree gets the highest accuracy of 98.88. The FNR is responsible for the majority of the models' inaccuracies. In other words, a large proportion of hazardous transactions are classified as benign by the detector (even using RF). In contrast, our framework's low FPR makes it helpful as a monitoring tool in situations in which the price of false positives is large, such as during testing or while pausing troublesome contracts in production for manual processing. The overall behavior of the models and performance evaluated based on accuracy , F1 score and Recall decision tree classifier is performing better(Fig-9).

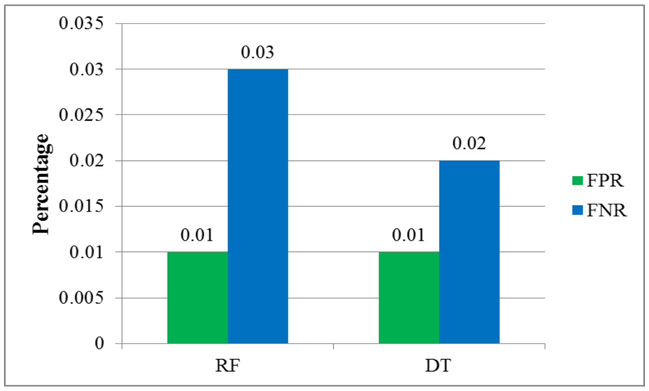


Figure-8 Using two alternative classification models, the average false positive and false negative rates for identifying susceptible transactions were calculated.

The contract sets we utilized to generate random transactions attempt to conceal their behaviour, as was previously noted. This step was taken to create a more accurate model and reduce bias[Fig-10,11].

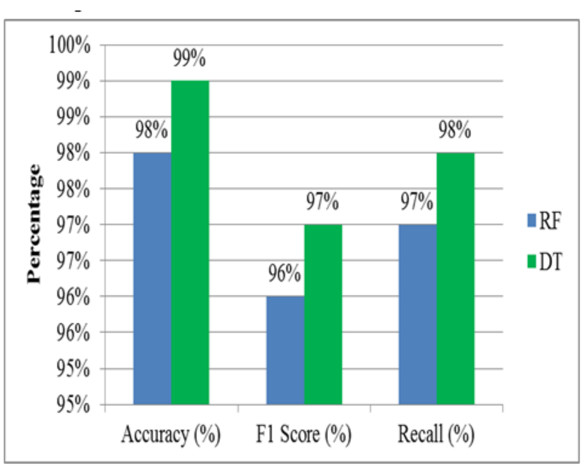


Figure-9 For Vulnerable Transactions with Two Different Classification models for Average Accuracy, F1 Score, and Recall.

In the heat map of confusion matrix performance of the classifier is checked by the variation of the test values and accuracy of the predication is almost same.

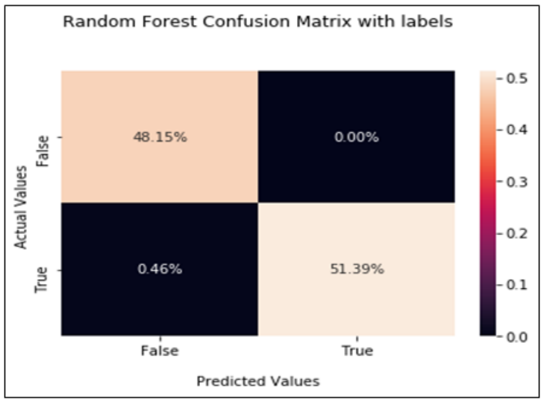


Figure-10 Overall Heatmap of Confusion Matrix for Random

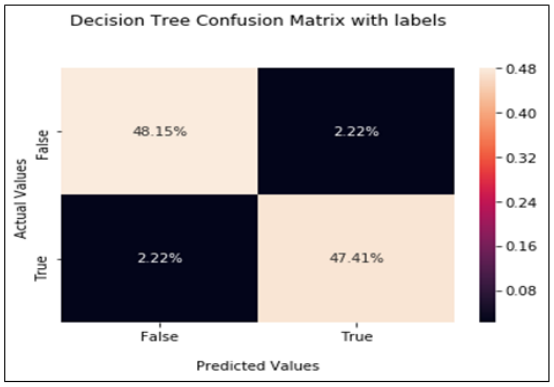


Figure-11 Overall Heatmap of Confusion Matrix for Decision Tree

Based on the TP and FP rate parameters also evaluated and performance of Decision tree is found to be more accurate as compared to the random forest.

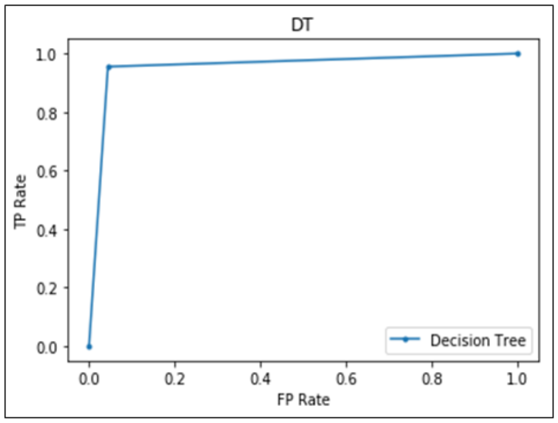


Figure-12 Performance Graph of Decision Tree on TP & FP rate

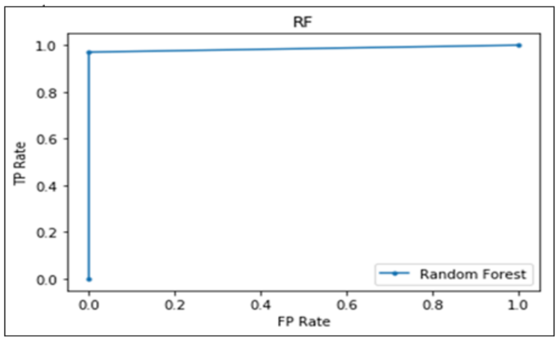


Figure-13 Performance Graph of Random Forest on TP & FP rate

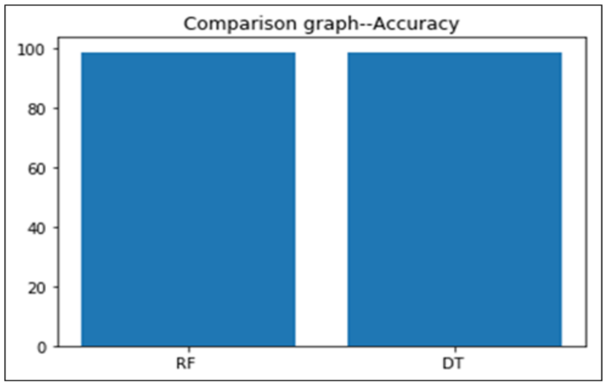


Figure-14 Overall Performance Graph for the Random Forest and Decision Tree

CONCLUSION & FUTURE WORK

In our work, we have reviewed all the possible tools available for vulnerability detection for smart contracts on Ethereum Blockchain and found very less number of tools there utilizing machine learning. In this paper, we also introduce the framework that a dynamic threat detection system for Ethereum smart contracts. Our methodology finds vulnerable smart contracts by categorizing damaging transactions inside a blockchain using machine learning over transactional information. We recorded 98 percent accuracy on the dataset with 540 transactions.

To further enhance this framework in the future, add functionality that may manufacture labeled transactions and establish both benign and malignant user contracts. For more accuracy in the results, investigate to identify more features for the machine learning model for better detection. To improve our scanner's functionality and assess more vulnerability, we will also investigate evaluating sequences of many transactions and applying multiple kinds of machine learning to the data.

REFERENCES

[1] Luu, L., Chu, D.-H. H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. Proceedings of the ACM Conference on Computer and Communications Security,24-28 October, 254–269. https://doi.org/10.1145/2976749.2978309

[2] Cryptocurrency Prices, Charts And Market Capitalizations | Coin Market Cap. (n.d.). Retrieved April 30, 2022, from https://coinmarketcap.com/

[3] Home | ethereum.org. (n.d.). Retrieved May 2, 2022, from https://ethereum.org/en/

[4] "DASP - TOP 10." https://dasp.co/ (accessed May 15, 2022).

[5] Zheng, P., Zheng, Z., & Dai, H.-N. (n.d.). X Block-ETH: Extracting and Exploring Blockchain Data From Ethereum. Retrieved June 7, 2022, from http://xblock.pro/dataset

[6] O. Lutz et al., "ESCORT: Ethereum Smart Contracts Vulnerability Detection using Deep Neural Network and Transfer Learning; ESCORT: Ethereum Smart Contracts Vulnerability Detection using Deep Neural Network and Transfer Learning."

[7] Basilio, J., 2021. Fitting Heavy Tail Distributions With Mixture Models. Stochastic Modelling and Computational Sciences, 1(1), pp.53-68.

[8] C. Dannen, "Introducing Ethereum and solidity: Foundations of cryptocurrency and blockchain programming for beginners," Introd. Ethereum Solidity Found. Cryptocurrency Blockchain Program. Beginners, pp. 1–185, Jan. 2017, doi: 10.1007/978-1-4842-2535-6/COVER.

[9] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contract Ward: Automated Vulnerability Detection Models for Ethereum Smart Contracts," IEEE Trans. Netw. Sci. Eng., vol. 8, no. 2, pp. 1133–1144, Jan. 2020, doi: 10.1109/TNSE.2020.2968505.

[10] "Reentrancy - Ethereum Smart Contract Best Practices." https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/ (accessed Jun. 11, 2022).

[11] Rameder, H., di Angelo, M., & Salzer, G. (2022). Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. Frontiers in Blockchain, 0, 2. https://doi.org/10.3389/FBLOC.2022.814977

[12] Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., & Lee, H. N. (2022). Ethereum Smart Contract Analysis Tools: A Systematic Review. IEEE Access. https://doi.org/10.1109/ACCESS.2022.3169902

[13] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, ''Smart Bugs: A framework to analyze solidity smart contracts,'' in Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng., Dec. 2020, pp. 1349–1352, doi: 10.1145/3324884.3415298

[14] Zhao, X. and Gilber, K., 2021. Estimating the Variance of Waiting Time in the Delivery of Health Care Services. International Journal Data Modelling and Knowledge Management, 6(2).

[15] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, ''Contract Guard: Defend Ethereum smart contract with embedded intrusion detection,'' Chin. J. Netw. Inf. Secur., vol. 6, no. 2, pp. 35–55, 2020, doi: 10.1109/TSC.2019.2949561.

[16] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, ''GASOL: as analysis and optimization for Ethereum smart contracts,'' in Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science), vol. 12079, A. Biere and D. Parker, Eds. Cham, Switzerland: Springer, 2020, doi: 10.1007/978-3- 030-45237-7_7

[17] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, ''S-gram: Towards semantic aware security auditing for Ethereum smart contracts,'' in Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng., Sep. 2018, pp. 814–819, doi: 10.1145/3238147.3240728.

[18] Feist, J., Grieco, G., & Groce, A. (2019). Slither: A Static Analysis Framework For Smart Contracts. https://doi.org/10.1109/WETSEB.2019.00008

[19] Wang, H., Liu, Y., Li, Y., Lin, S.-W., Artho, C., Ma, L., & Liu, Y. (n.d.). Oracle-Supported Dynamic Exploit Generation for Smart Contracts.

[20] GitHub - ethereum/web3.py: A python interface for interacting with the Ethereum blockchain and ecosystem. (n.d.). Retrieved June 28, 2022, from https://github.com/ethereum/web3.py

[21] Yazici, H., 2020. Knowledge Sharing Antecedents in Buyer-Seller Chains. Chinese Journal of Decision Sciences, 2(1).

[22] "pandas - Python Data Analysis Library." https://pandas.pydata.org/ (accessed Jun. 24, 2022).

[23] Ethereum (ETH) Blockchain Explorer. (n.d.). Retrieved June 30, 2022, from https://etherscan.io/

[24] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (n.d.). Vul Dee Pecker: A Deep Learning-Based System for Vulnerability Detection. https://doi.org/10.14722/ndss.2018.23158

[25] Sri Handika Utami et.al., 2022. Fintech Lending in Indonesia: A Sentiment Analysis, Topic Modelling, and Social Network Analysis using Twitter Data. International Journal of Applied Engineering and Technology , 4(1)