Advances in Knowledge-Based Systems, Data Science, and Cybersecurity 2 (2) 215-235 Received 02-04-2025; Accepted 16-05-2025; Published 23-05-2025

Smart Contracts- Vulnerabilities, CodeLlama Usage and Gas Driven Detection

Natan Katz natan.katz@gmail.com
Kehilat Wrsaw 15B Tel Aviv

Corresponding Author: Natan Katz

Abstract

Smart contracts are a major tool in Ethereum transactions. Therefore, hackers can exploit them by generating malicious code and using their vulnerabilities to perform malicious transactions. This paper presents two successful approaches for detecting malicious contracts: one uses opcode and relies on GPT2, and the other uses the Solidity source and a LORA fine-tuned CodeLlama. Finally, we present an XGBOOST model that combines gas properties and Hexadecimal signatures for detecting malicious transactions. This approach relies on early assumptions that maliciousness is manifested by the uncommon usage of the contracts' functions and the effort to pursue the transaction.

Keywords: LLM, Fine-tuning, CodeLlama, Smart contracts, Ethereum, ABI, Bayesian networks, Deep Learning, Causal Inference.

1. INTRODUCTION

Machine learning **ML** has become a tool in various technological domains. The appearance of tools such as **ChatGPT** [1] and **DALL-E** [2] made ML accessible to an enormous number of users. However, ML learning influences even more aspects than the common generative approaches: we can see the usage of ML in the medical world, in financial, and even in less quantitative disciplines such as archaeology and history. It intrigues nearly everyone that when we provide data samples to software, we enable it to extract novel insights. In parallel to the rise of ML, cybersecurity has gained more focus lately. The increase in network traffic, particularly in money transactions and the usage of financial information, made most of us vulnerable to malicious web activities. One may expect that cyber and ML will constantly shake hands. Indeed, we see a massive presence of ML in cybersecurity applications. However, creating an ML solution for cyber is often a challenging task due to the following reasons:

- **Imbalanced data-** Most of the traffic in cyber is benign: we seldom have traffic that contains more than a single percent of malicious data. Since imbalanced data is one of the classical obstacles in ML, we may suffer from a tall hurdle.
- **Temporality-** In most of the common ML applications, such as image detection or sentiment analysis [3], we can confidently assume that we sample the data from a fixed distribution.

In cybersecurity, the data is often temporal since the attacker and the defender experience an evolution.

- **Labeling-** Unlike vision or text, where it is easy to label the data in advance (every junior high pupil can say if the image is of a dog or a cat), labeling cyber data requires expert knowledge.

These obstacles impose an effort in nearly every cyber ML project. Performing ML projects on blockchain may require even more effort since it may present novel challenges. This paper describes potential cyber threats in the Blockchain and ML methods to solve them.

2. SMART CONTRACTS

The following section introduces the concept of smart contracts and the potential problems we may encounter when using them that they may rise.

2.1 What is Ethereum?

In 2013, Vitalik Buterin had an idea to improve Bitcoin [4–11]. Rather than the block including only transactions, Vitalik wanted it to include sources of code. His motivation was the need to use the blockchain properties to develop decentralized applications, **Dapps**, and through creating the world's strongest supercomputer, [5, 7]. However, there was a tall hurdle that Vitalik had to face: Bitcoin used a scripting language **Bitcoin Script** [5, 7], that wasn't adequate for Vitalik's goal: This language wasn't **Turing Complete**, namely, there are logical problems that one cannot code using it. To achieve Turing completeness, one had to enrich it with loops, a programming feature that **Bitcoin Script** didn't support. The developers wanted to prevent tedious effort, such as infinite loops, that can significantly slow the network. Vitalik wanted to enrich **Bitcoin Script** with loops. He didn't get his request. Vitalik moved forward with his idea and established a new blockchain-based network: Ethereum. While Bitcoin acts merely as a cryptocurrency, Ethereum indeed followed Vitalik's vision in two aspects:

- It became a platform for developers to create their applications and for users to run these applications [7, 8, 12].
- In contrast to Bitcoin, Ethereum's blocks contain, in addition to transactions, source codes as well. One can run these codes on the network's nodes.

These two achievements allow Ethereum to decentralize many computing tasks and, as mentioned above, develop many **dApps**. We can summarize with the words of Dr. Gavin Wood, one of the Ethereum founders: **Bitcoin Script** is first and foremost a currency; this is one particular application of a blockchain. However, it is far from the only application. To take a past example of a similar situation, e-mail is a particular use of the internet, and undoubtedly helped popularize it, but there are additional Internet usages [7]. Ethereum allows more capabilities and requires a more clever environment. In the following sections, we will describe these properties.

2.2 Ethereum Virtual Machine

We discussed in the previous section the need for improving blockchains with source code. This improvement brings two threats[5, 6]:

- Infinite loops
- Public accessibility

In this section, we discussed the latter. A blockchain user has the entire chain on their private computer. If we place the sources themselves on the blocks, we may suffer from the following risks:

- The source codes can access the private drivers of the users.
- If it is plausible to add sources to the blocks, it is easy to deploy viruses

To overcome these risks [5, 7], every participant in Ethereum receives a virtual machine that separates their Ethereum activities from the personal drivers: Ethereum Virtual Machine **EVM**. The developers upload their sources to the chain, and **EVM** executes these sources. The separation from the private drives enhances security.

2.3 What is a Smart Contract?

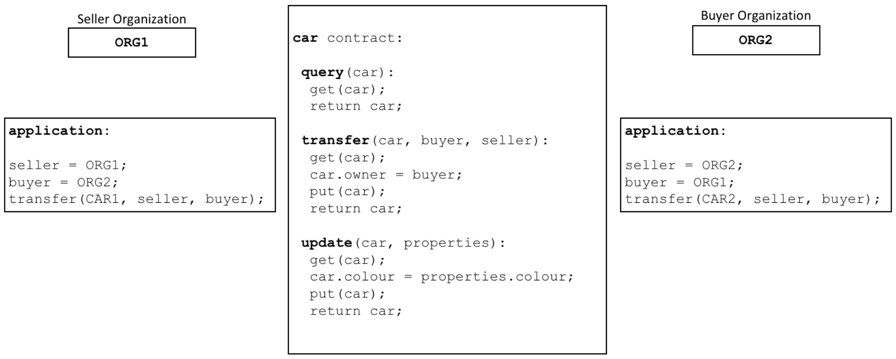
In a nutshell, smart contracts are merely contracts. The main differences between these contracts and the common ”real world” contracts are [5, 6, 12–14]:

- Blockchains use them
- They are computer programs.

The concept of source-codes allows **Dapps** to use them as their back-end for managing the API and the interaction with the blockchain itself. Why do we use the term **”contract”**? I will refer to a great post [12], where the writer uses the idea carved on stone contracts. These contracts provide a high level of trust. In a more modern example [12], one can consider a vending machine: It will output a Diet Coke if you provide a certain amount of money and press the button. One can see in FIGURE 1, that this procedure can be programmed [12]. We can create more sophisticated contracts [15], as one can see in FIGURE 2.

```
if money received == $2.50 && the button pressed is "Diet Coke" :
     release Diet__Coke
|
```

Figure 1: Vending Machine Contract



```
application:

seller = ORG1;
buyer = ORG2;
transfer(CAR1, seller, buyer);
```

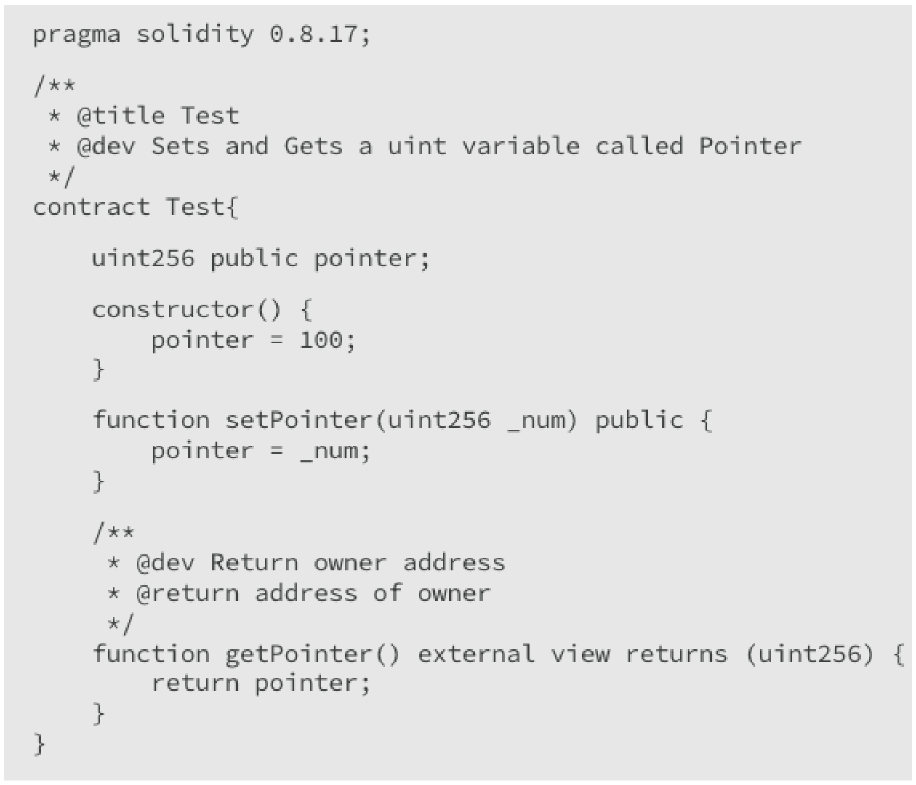
```
car contract:

 query(car):
  get(car);
  return car;

 transfer(car, buyer, seller):
  get(car);
  car.owner = buyer;
  put(car);
  return car;

 update(car, properties):
  get(car);
  car.colour = properties.colour;
  put(car);
  return car;
```

```
application:

seller = ORG2;
buyer = ORG1;
transfer(CAR2, seller, buyer);
```

Figure 2: Selling a car using a smart contract

2.3.1 Smart contract compilation

We use a special language program for smart contracts: **Solidity**, [5, 10]. An example of a Solidity code [16], can be seen in 3, and the entire compilation process[16], in 4 We can see



```
pragma solidity 0.8.17;

/**
 * @title Test
 * @dev Sets and Gets a uint variable called Pointer
 */
contract Test{

    uint256 public pointer;

    constructor() {
        pointer = 100;
    }

    function setPointer(uint256 _num) public {
        pointer = _num;
    }

    /**
     * @dev Return owner address
     * @return address of owner
     */
    function getPointer() external view returns (uint256) {
        return pointer;
    }
}
```

Figure 3: Solidity source example

that there are two outcomes for the output of the compilation process :

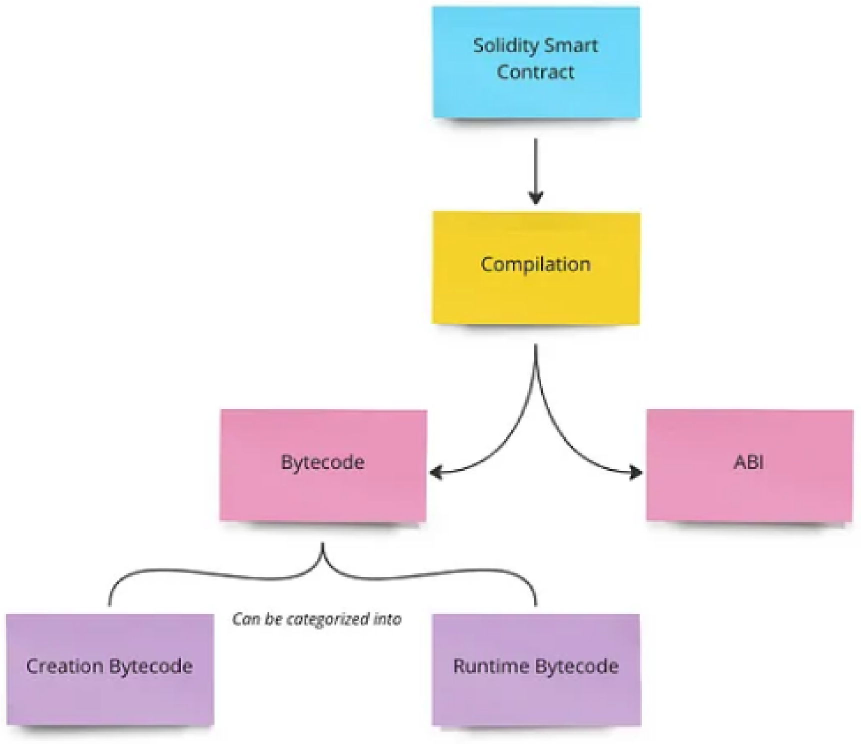


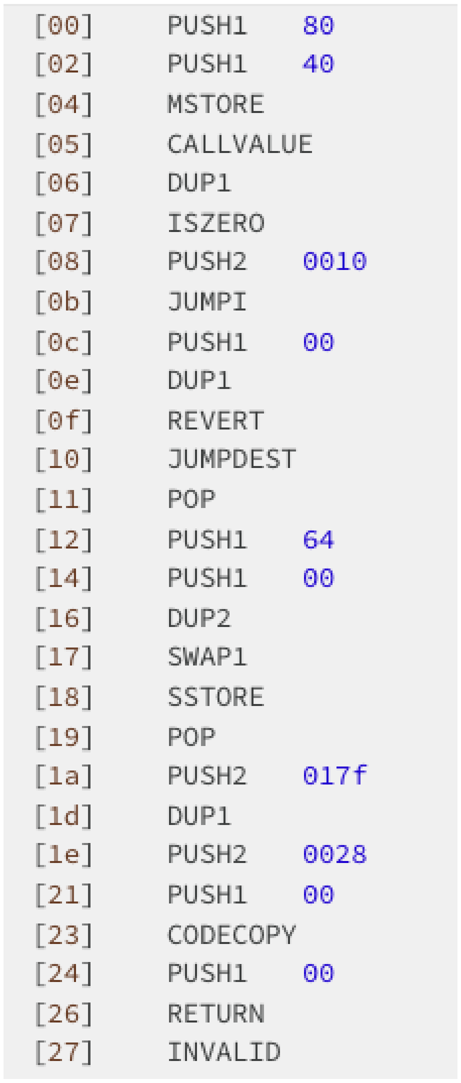
Figure 4: Smart contract's compilation chain

- Application Binary Interface **ABI** This interface clarifies which functions and which parameters the contract uses [10, 16, 17, 17]
- Bytecode - A hexadecimal string that encodes the source code. The developer uploads the bytecode to the EVM. The EVM extracts the opcode instructions [10, 16]. An example opcode can be seen [16], in 5 respectively.

2.3.2 Gas and infinite loops

We described the potential problem that Ethereum may encounter with heavy calculations in general and infinite loops in particular. In this section, we will discuss how Ethereum handles this obstacle [5, 6, 10]. Every source code is an ordered collection of commands. Thus, Ethereum uses the concept of **Gas** [5]. The idea is that a user pays for every requested computation command on the network. Ethereum publishes a pay list that assigns a gas fee for each command. This approach provides two significant benefits:

- There are no infinite loops- A user can run a contract only if she can budget the required computational resources
- Code is written efficiently



```
[00]    PUSH1   80
[02]    PUSH1   40
[04]    MSTORE
[05]    CALLVALUE
[06]    DUP1
[07]    ISZERO
[08]    PUSH2   0010
[0b]    JUMPI
[0c]    PUSH1   00
[0e]    DUP1
[0f]    REVERT
[10]    JUMPDEST
[11]    POP
[12]    PUSH1   64
[14]    PUSH1   00
[16]    DUP2
[17]    SWAP1
[18]    SSTORE
[19]    POP
[1a]    PUSH2   017f
[1d]    DUP1
[1e]    PUSH2   0028
[21]    PUSH1   00
[23]    CODECOPY
[24]    PUSH1   00
[26]    RETURN
[27]    INVALID
```

Figure 5: Opcode’s Insurrections

Gas is, therefore, a mandatory component for achieving Turing completeness in a blockchain. This concept emphasizes that Ethereum (in contrast to Bitcoin) was developed as a computational platform, not merely a cryptocurrency.

3. MACHINE LEARNING

Machine learning **ML** is not a new discipline. Researchers used tools such as decision trees and probabilistic modeling for decades [18, 19]. The usage of ML has been boosted during the last years by the Deep learning **DL** revolution that lately reached a highlight with the launch of ChatGPT [1]. While tools like Random Forest [18–20], and XGBoost [19, 21, 22], are known to most researchers, in this section, I will describe some of the deep learning tools that we used. In this paper, we utilize transformer textual models [23, 24], and autoencoders [19].

3.1 Bayesian Network

Causal inference is a rising research domain [25–27]. It offers promising results in various disciplines of applied science. Some of its tools can enhance the methodologies by which researchers approach data [25, 26]. A core object in causal inference is the Directed Acyclic Graph **DAG**, [25, 26]. It represents the actual relations between variables and allows us to perform causal analysis 6. Moreover, the power of DAG is in uncovering latent rela-

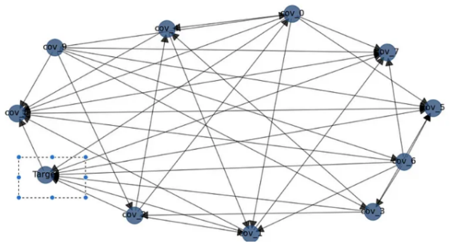
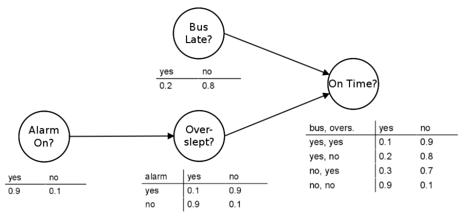


Figure 6: Directed Acyclic Graph - AG

tions between the raw variables and improving the model's explainability [28]. However, in real-world scenarios, the directions of the arcs are often unknown; thus, researchers use probabilistic tools to approximate them. Bayesian networks are such a tool [28–30]: they denoise the data and uncover the mutual influence between the observed variables (FIGURE 7). In the following sections of this paper, we will describe Ethereum ML research that we



yes	no
0.2	0.8

yes	no
0.9	0.1

alarm	yes	no
yes	0.1	0.9
no	0.9	0.1

bus, overs.	yes	no
yes, yes	0.1	0.9
yes, no	0.2	0.8
no, yes	0.3	0.7
no, no	0.9	0.1

Figure 7: Bayesian Network [31]

did using the described tools for solving some of the problems that we discussed in section

4. REENTRANCY ATTACK-REAL WORLD EXAMPLE

We saw in that indeed smart contracts are a powerful tool. Nevertheless, as with any other technology, it comes with its "novel" risks. Smart contracts are used in Ethereum transactions; hence, their vulnerabilities can be exploited in these transactions. Since it is merely a source code, we can suspect that if we don't develop it properly, we may suffer drawbacks [8, 10, 32–34]. One of the most common attacks on smart contracts is the **reentrancy attack**. In such an attack, a fund transaction takes place. Rather than updating the paying account and transferring the funds in the following step, the contract transfers the funds before. As a result, one can call the transfer function infinitely many times and transfer the entire money of the victim. A graphical scenario of **reentrancy attack** is presented in FIGURE. 8. This scheme is presented in [32, 33].

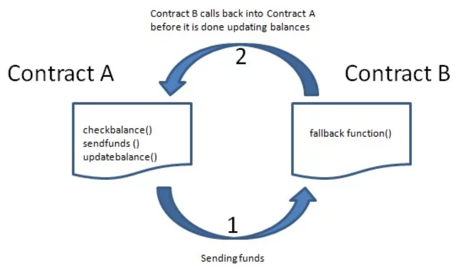
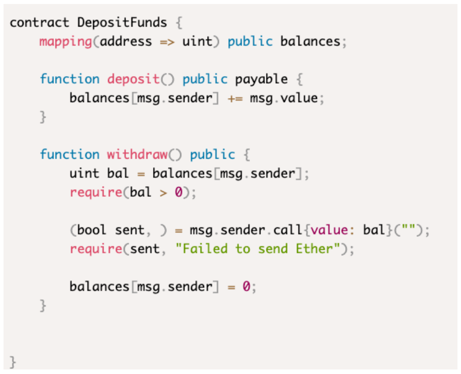


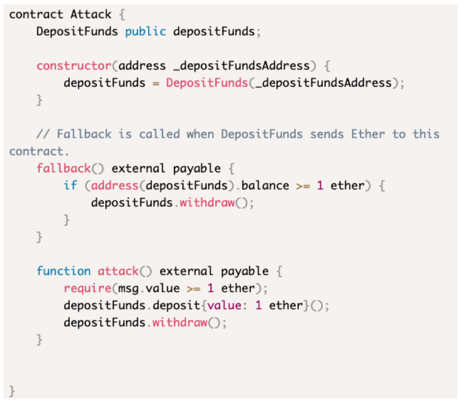
Figure 8: Reentrancy Scheme

We present a Solidity implementation of the reentrancy attack in FIGURES 9 and 10, [32]. The most famous **reentrancy attack** is the **DAO Attack**. It took place in 2017. 60



```
contract DepositFunds {
    mapping(address => uint) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");

        balances[msg.sender] = 0;
    }


}
```

Figure 9: Deposit Function



```
contract Attack {
    DepositFunds public depositFunds;

    constructor(address _depositFundsAddress) {
        depositFunds = DepositFunds(_depositFundsAddress);
    }

    // Fallback is called when DepositFunds sends Ether to this
contract.
    fallback() external payable {
        if (address(depositFunds).balance >= 1 ether) {
            depositFunds.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        depositFunds.deposit{value: 1 ether}();
        depositFunds.withdraw();
    }


}
```

Figure 10: Reentrancy Attack function

million dollars were stolen. Moreover, the primary outcome of this attack was a hard fork of Ethereum, resulting in two separate communities. One can learn about the impacts of this attack on Ethereum [5, 6, 8]. DAO was followed by additional attacks, such as [32] :

- Uniswap/Lendf.Me hacks (April 2020) – 25 million dollars, attacked by a hacker using reentrancy.
- The BurgerSwap hack (May 2021) – More than seven million dollars were attacked, due to a reentrancy exploit.

5. RESEARCH METHODOLOGY

We present three ML studies that focus on Ethereum:

- Malicious detection based on smart contract opcode using GPT2 [35–37].
- Malicious detection based on smart contract Solidity source-using CodeLlama [38–40].
- Malicious transaction detection, which is based mainly on gas-driven variables and uses classical ML tools (mainly XGBOOST).

In the last one, we assumed that two:

- The user is willing to pay a high amount of gas to place the transaction as quickly as possible
- The user may choose different approaches to use common smart contract functions

5.1 Malicious Data

Since we develop detection models, we need labeled data for both populations (e.g., benign and malicious). While there are many benign smart contract source codes ([41]), Malicious sources are not always available:

- People don't leave traces such as Solidity sources or their ABI when they act maliciously.
- With the absence of prior knowledge, one can hardly label a contract as malicious.
- Transactions are rarely labeled as malicious

These obstacles reveal a flaw in Ethereum design: A smart contract developer must upload only the Opcode. Solidity source and ABI are not mandatory and don't require explicit community support. This causes difficulty in collecting data, but moreover, it undermines the network's decentralized ethos. Nonetheless, to overcome this ”data issue,” we used common websites such as [42, 43]. that indicate when contracts are suspicious, and decompiled their Opcode. The training results that we describe below indicate that our model generalized the information well,

6. THE OPCODE PROJECT

6.1 Smart Contracts' Detection

Malicious detection of smart contracts has already been done [41]. This approach is beneficial:

- The code is easy to disassemble from the bytecode
- The commands are human-readable, thus LLMs can handle them
- We can detect statistical patterns and therefore gain explainability [41].

To achieve our goal, we used a text generation model- GPT2 [36, 37]. We had 10k benign smart contracts and about 800 malicious ones. Namely, we suffer from an imbalanced dataset. Prior to fine-tuning the model, we measure its performance 11 One can see that the detection

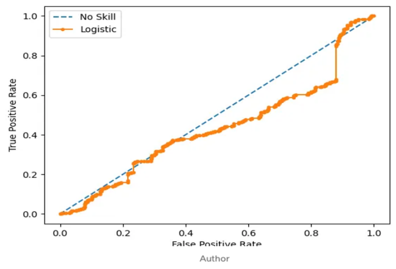


Figure 11: GPT2 False Positive Vs. recall

rate is nearly random. We use **GPT2** as our initial model and fine-tune it. During our early trials, we simply flattened the contracts to obtain poor results. We realized that truncating the prefix and restricting the length to 600 commands (with padding if needed) improved our performance, as shown in FIGURE12.

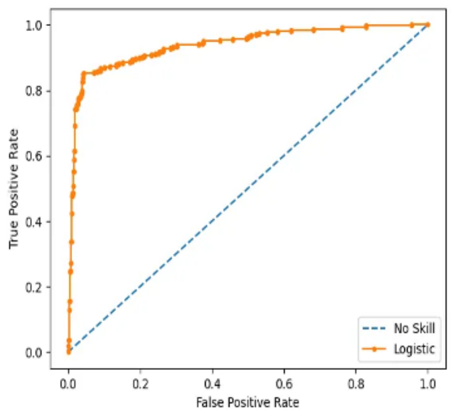
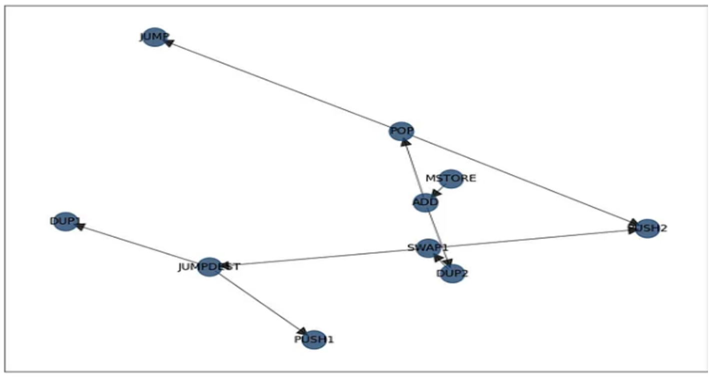
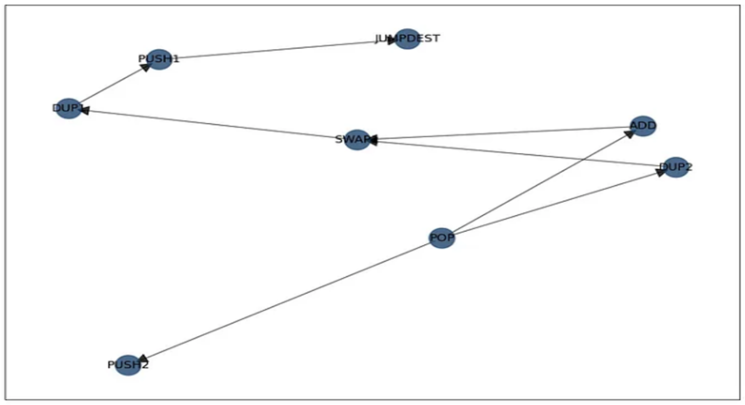


Figure 12: Trained GPT2 False Positive Vs. recall



(a) Malicious DAG



(b) Benign DAG

Figure 13: DAGs Comparison

6.2 Opcode Graphs

In addition to a good prediction model, ML Cyber projects often require a coherent explainability mechanism. While we don't have features, the Opcode commands can be informative. For studying patterns in these commands, we used Python's bnlearn [30]. to generate characteristic DAGs for each of the contract populations. These DAGs are presented in FIGURE 13.

7. THE SOLIDITY PROJECT

Motivated by the Opcode project, we further study a Solidity detection model. As we explained above, malicious source codes are less available than Opcode. To overcome this absence, we used websites such as Chainabuse [42]. and recompiled Opcode sources using a web decompiler [44]. Our data includes about 100k benign contracts and about 2000 malicious contracts with their (recompiled) Solidity sources. We used the Huggingface database as our source of benign contracts [45].

7.1 Our KPIs

There are several common KPIs that data scientists use for assessing model performance:

- **Accuracy-** A metric that measures how accurately the model predicted. It is a beneficial metric when the data is uniform. However, using accuracy as a metric is less informative when one class (benign) has a much higher prevalence than the other.
- **Precision-** This metric measures the probability that a class detection is correct. Namely, the probability that a contract is benign given that the model declared ”benign”.
- **Recall-** This metric measures the probability of detecting a certain class.
- **F1** The harmonic average of precision and recall.
- **confusion Matrix**

Actual \ Predicted	Positive	Negative
Positive	TP	FN
Negative	FP	TN

Table 1: Confusion matrix: T-True, F-False, P-Positive, N-Negative

Metrics like accuracy, precision, and F1-score require selecting a specific decision threshold. While this provides insight into model performance at that particular point, it discards valuable information about how the model behaves across other thresholds. In contrast, the ROC curve captures the model’s performance across the full range of thresholds, offering a more comprehensive and threshold-independent evaluation. A reader who is interested in having a benchmark needs to consider the following:

- Metrics such as F1 precision require a clear, coherent threshold.
- In common applications such as image classification or sentiment analysis, the model decides upon the maximum probability criterion. One must not follow this criterion in asymmetric applications, such as malicious detection

Finally, ROC indicates the delta between the classes’ intrinsic distribution [46], Therefore, we present the ROC curves, and the system user can decide about their preferred threshold (namely, which class wins in a single inference step), as well as the working point.

7.2 CodeLlama

We aim to train a model that is fed by Solidity sources and outputs a binary decision. Llama of Meta [38, 39]. offers code generative models, specifically **CodeLlama**, which was trained on Java and Python sources [40]. For modifying such a model into a classifier, we require the following:

- Modifying the architecture by adding a proper matrix
- Fine-tune the model

The former is described in [47]. The latter is required since the model wasn't trained on Solidity sources or malicious code. Our focus is therefore on classifying Solidity contracts. Early trials achieved about 70 percent accuracy. However, this is a 7B model with very little data. Since we cannot increase our dataset, we can decrease the model.

7.3 The Solidity Generator

We found a 500M Solidity generator model [48], which is based on **BERT** [49]. This model was trained on [50]. We followed the steps of [47]. to modify its architecture. The results are presented in 14 In the following graph, the X-axis represents precision, and the Y-axis,

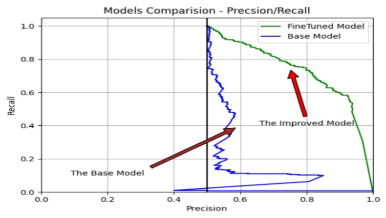


Figure 14: The Solidity Classification Model

the recall. The base model is the model before our fine-tuning. We can see that on the one hand, the results are not outstanding due to the absence of data. On the other hand, we can observe the improvement due to the fine-tuning process (Simply, note the gaps between the curves, or measure actual points).

7.3.1 Discussion

The performances indicate that our data is ”trainable”, namely, a DL tool can detect statistically whether a smart contract is malicious or not. Moreover, the precision-recall graph shows that the knowledge was scattered across both classes. However, to achieve commercial performances, we must increase our data, particularly malicious contract sources. Testing different code models may improve the results. Since the concern in Ethereum is transactions, detecting a malicious contract is not only an objective but also a supporting tool. In the next section, we discuss transaction detections where smart contracts can be used.

8. ML STUDY OF MALICIOUS TRANSACTIONS

In this section, we present a transaction detection model. We recall that we restrict ourselves to transactions in which the receiver is a smart contract. The model's features are based on the structure of **gettransaction** and **gettransactionreceipt** by the API of Python's web3 package [51].

8.1 Data

As in the previous section, we need labeled data of both malicious and begin transactions. For this purpose, we used websites such as **de-fi, chainabuse** [43, 52]. We collected about 15K malicious transactions. We sampled benign transactions from two sources:

- ”Modern” blocks.
- Blocks that contain malicious transactions (to overcome temporality)

8.1.1 Our features

Our research focuses on variables that describe aspects of the gas bid-ask and the input functions. We briefly describe some of the variables:

- **effectiveGasPrice** -The effective gas price that was paid
- **cumulativeGasUsed** - The amount of gas in the block until the relevant transaction,
- **to** - Obvious field . Here, we need to verify that it is a contract.
- **logs** The number of logs that appeared during the execution of the smart contract.
- **gasUsed/gasPrice** The gas was used in the transaction and its cost in wei.
- **type** It indicates the mechanism of Ethereum that influences the gas pricing (EIP-1559.
- **gas** -Maximum amount of gas that the sender is willing to pay
- **maxFeePerGas**- Amount of gas that one is willing to pay for computational resources
- **maxPriorityFeePerGas**- Maximum tip one is willing to pay

The model's input features are either these variables or their aggregators. The motivation to focus on gas-driven variables is derived from the assumption that a potential culprit will offer a high amount of gas to execute their transactions.

8.2 The Model

Since we are using tabular data, the common tool is XGBOOST [21, 22] We trained the XGBOOST model and got the following ROC: In addition, we present the feature importance of the xgboost model: In this graph, the width of the bar indicates the importance of the variable in the model's decision, thus it is a coherent explainability metric. One can see that the gas aggregators are indeed achieving the greatest importance score, as we pre-assumed.

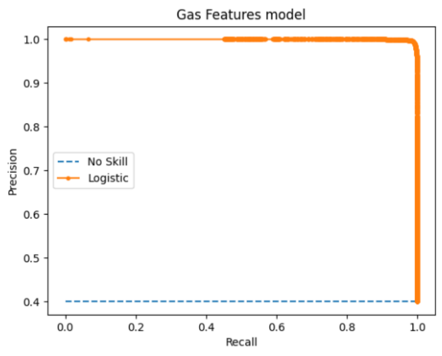


Figure 15: ROC curve of the xgboost model

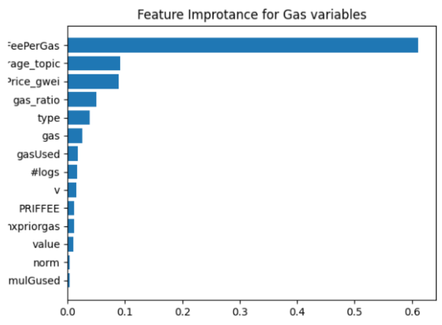


Figure 16: Feature Importance of our model

8.3 The 4bytes Analysis

Every transaction includes a field called **input** which is a hexadecimal signature that indicates the functions of the smart contract and their inputs. There is a website **4bytes** [53], that provides a map from many of these hexadecimal signatures to the actual smart contracts' functions. We wish to extract information about these functions and whether they influence malicious transactions. We developed a mapping from the Hexa signatures to the functions. We denote the latter **hexdic**. The following graph represents the distribution of signatures: We can see that most of the known signatures appear only in benign transactions. **But there**

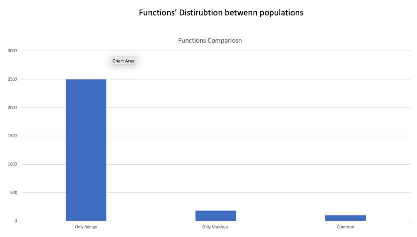
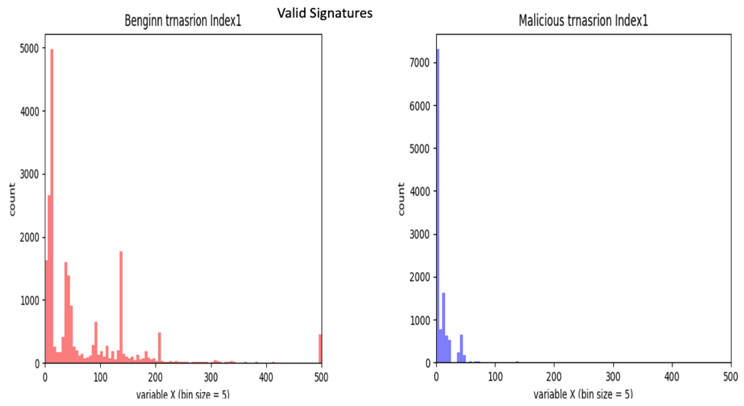
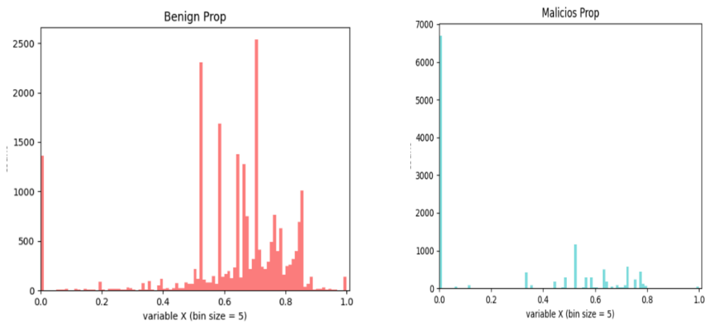


Figure 17: Signatures' histogram

are about 100 signatures that never appear in our 60k benign transactions. Hence, these signatures are likely to be good malicious indicators.



(a) Valid Sig



(b) Prop valid

Figure 18: Valid signature Distribution

The next feature that we studied was the functions' input length

We can see that these distributions are different; hence, the input length can be introduced as a feature to the model The main point is not what distribution parameters are, but the idea that these distributions are not from the same class. A future study can rigorously model each of these distributions and provide their parameter tuples to the model. We trained the model with our new features and obtained the following results: We add the feature importance as well: It can be seen that the results are slightly better. We can explain it by the fact that many transactions have no ABI. Nonetheless, it emphasizes the Ethereuem's flaw, which we discussed that developers are free from presenting their sources

8.4 Deep learning Early Study

As we discussed in the previous sections, it is often difficult to have malicious contracts. Thus, a different approach can be studied: We use an auto-encoder to train a representation of the benign contract and consider the malicious contracts as anomalies. Following such

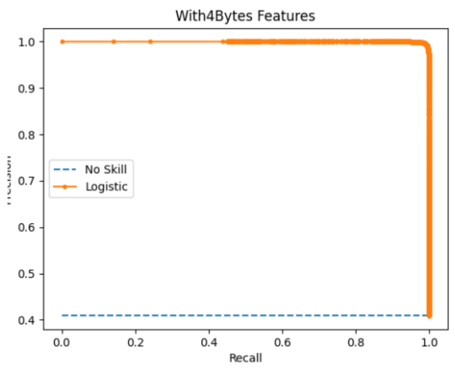


Figure 19: Model's ROC with 4bytes

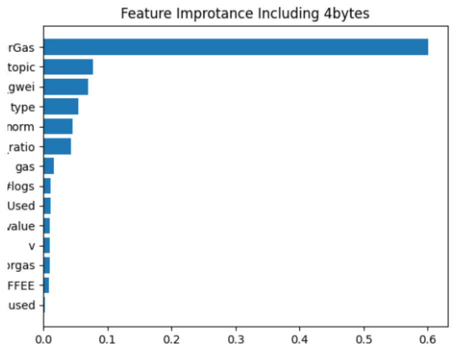


Figure 20: Model's Features' importance with 4bytes

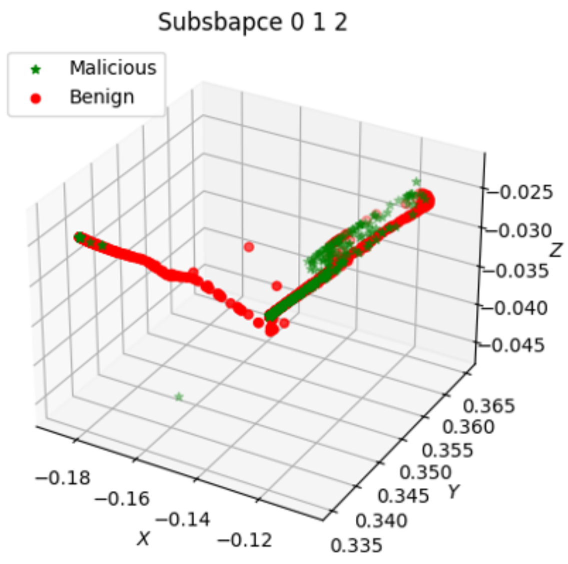
training, we projected our data on the three components of the PCA benign subspace [54], as shown in FIGURE21 :

We can observe that a separation between the malicious and the benign takes place. A further step is to learn the distribution of coefficients from benign contracts. This enables us to assign a p-value to each malicious contract, quantifying its rarity under the benign distribution. One can use a common anomaly detection tool such as Isolation-forest [55].

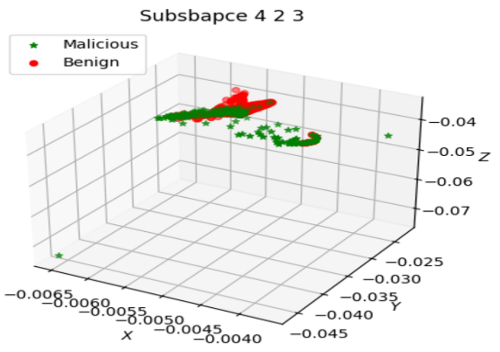
9. SUMMARY AND FUTURE STEPS

We have presented several projects that use ML to handle maliciousness in Ethereum. This tool seems to be beneficial on various levels. Following our work, the next natural steps need to focus on two directions:

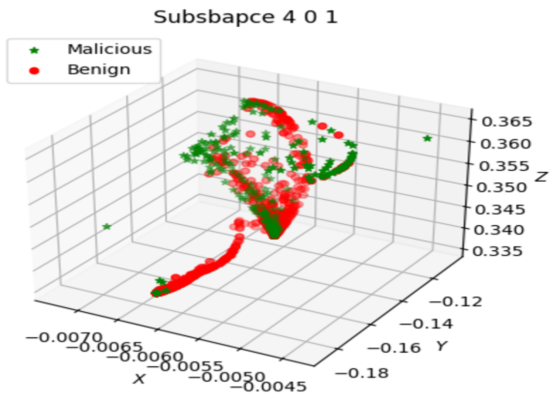
- Improving the data collection methodology. In particular, malicious contracts.
- Finding novel methods to exploit LLMs to solve Web3 tasks in general and blockchain security in particular,



(a) Components 0 1 2



(b) Components 2 3 4



(c) Components 0 1 4

Figure 21: Transactions Projected on PCA

- Improving the transactions detection models
- Perform a better usage of the 4-byte information (using anomaly detection and probabilistic methods)

The code of this project can be found here [56].

10. ACKNOWLEDGEMENT

I would like to thank Amit Levin for bringing forth the initial idea that inspired this work. My gratitude goes to Roma Zaikin, Dikla Barda, and Oded Vanunu for their valuable clarifications on key topics. I am especially grateful to Avishay Shraga for his in-depth, profound discussions and innovative ideas, which greatly enriched this research. Finally, I would like to thank our team at LuminAI Security for their tireless efforts.

References

[1] *https://chat.openai.com/*

[2] *https://openai.com/dall-e-2*

[3] *https://en.wikipedia.org/wiki/Sentiment_analysis*

[4] *https://etherscan.io/*

[5] *https://www.udemy.com/course/build-your-blockchain-az*

[6] *V. Buterin. DAOs, DACs, DAs and More: An Incomplete Terminology Guide. 2014. Available at : https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide*

[7] *A. Moskov . What is Ethereum - The Ultimate Beginner's Guide. 2017. https://coincentral.com/what-is-ethereum-the-ultimate-beginners-guide*

[8] *M. Leising .The Ether Thief 2017. A vailable at :https://www.bloomberg.com/features/2017-the-ether-thief*

[9] *https://www.freecodecamp.org/news/smart-contracts-for-dummies*

[10] *Vanunu, Zaikin, Barda . Cyber and Hacking in the Worlds of Blockchain and Crypto. 2023*

[11] *https://ethereum.org/en/what-is-ethereum/*

[12] *N. Custodio .Smart Contract for Dummies. 2017. Available at: https://www.freecodecamp.org/news/smart-contracts-for-dummies-a1ba1e0b9575/*

[13] *https://cryptorunner.com/what-is-ethereum/*

[14] *https://ethereum.org/se/developers/docs/smart-contracts/*

[15] *https://coinloan.io/blog/smart-contract-platforms-overview*

[16] *https://medium.com/coinmonks/evm-part-ii-the-journey-of-smart-contracts-from-solidity-code-to-bytecode*

[17] *Louis Abraham's Home Page, 2022. Calling a Contract Without ABI on Ethereum https://louisabraham.github.io/articles/no-abi*

[18] *R. O. Duda , P. Hart , D. G. Stork Pattern Classification. Wiley, 2000.*

[19] *https://cs229.stanford.edu/*

[20] *https://towardsdatascience.com/understanding-random-forest*

[21] *https://xgboost.readthedocs.io/en/stable/*

[22] *https://xgboost.readthedocs.io/en/stable/tutorials/model.html*

[23] *Mikolov T, Karafiát M, Burget L, Cernocký J, Khudanpur S. Recurrent Neural Network Based Language Model. InInterspeech 2010;2:1045-1048.*

[24] *Sutskever I, Vinyals O, Le QV. Sequence to Sequence Learning With Neural Networks. 2014 https://arxiv.org/pdf/1409.3215*

[25] *A. Molak. Causal Inference and Discovery in Python: Unlock the Secrets of Modern Causal Machine Learning With DoWhy, EconML, PyTorch and More. Packt Publishing.2023*

[26] *https://www.youtube.com/c/BradyNealCausalInference*

[27] *https://medium.com/p/4dc706680294*

[28] *https://medium.com/towards-data-science/explainability-using-bayesian-networks-4dc706680294*

[29] *https://www.bnlearn.com/examples/*

[30] *https://pypi.org/project/bnlearn/*

[31] *https://www.uib.no/en/rg/ml/119695/bayesian-networks*

[32] *https://hackernoon.com/hack-solidity-reentrancy-attack*

[33] *https://cryptomarketpool.com/reentrancy-attack-in-a-solidity-smart-contract/*

[34] *https://medium.com/immunefi/the-ultimate-guide-to-reentrancy-19526f105ac*

[35] *Radford A, Narasimhan K, Salimans T, Sutskever I. Improving Language Understanding by Generative Pre-Training 2018.*

[36] *Radford A, Wu J, Child R, Luan D, Amodei D. Language Models Are Unsupervised Multitask Learners. OpenAI blog. 2019;18:9.*

[37] *https://huggingface.co*

[38] *https://ai.meta.com/llama/*

[39] *huggingface.co/docs/transformers/model_doc/llama2*

[40] *https://huggingface.co/docs/transformers/main/model_doc/code_llama*

[41] *https://forta.org/blog/how-fortas-predictive-ml-models-detect-attacks-before-exploitation/*

[42] *https://www.chainabuse.com/*

[43] *https://de.fi/*

[44] *https://ethervm.io/decompile*

[45] *https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts*

[46] *Natan Katz and Uri Utai. Parametric PDF for Goodness of Fit. Advances in Artificial Intelligence and Machine Learning 2023;3:47.*

[47] *N Katz, CodeLlama FineTuning for Classification, Available at :https://medium.com/@natan-katz/codellama-classification-finetuning-28fa5546f64f*

[48] *https://huggingface.co/ckandemir/solidity-generator*

[49] *Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2019. Arxiv preprint : https://arxiv.org/pdf/1810.04805*

[50] *https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts*

[51] *https://web3py.readthedocs.io/en/stable/*

[52] *https://www.chainabuse.com/*

[53] *https://www.4byte.directory*

[54] *https://builtin.com/data-science/step-step-explanation-principal-component-analysis*

[55] *https://scikit- learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html*

[56] *https://github.com/natank1/finetunecodellama2binary/tree/main*