Smart Contracts- Vulnerabilities, CodeLlama Usage and Gas driven Detection

Natan Katz

December 2023

Abstract

Smart contracts are a major tool in Ethereum transactions. Therefore hackers can exploit them by adding code vulnerabilities to their sources and using these vulnerabilities for performing 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 Hexa-decimal 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, cyber security 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 cyber-security applications. However, creating an ML solution to 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 cyber security, 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 [6, 5, 11, 12, 14, 16]. Rather than the block including only transactions, Vitalik wanted them 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, [11]. However, there was a tall hurdle that Vitalik had to face: Bitcoin used a scripting language Bitcoin Script [11] that wasn't adequate for Vitalik's goal: This language wasn't Turing Compete, 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 efforts 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 crypto-currency, 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 [5, 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 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 popularise it, but there are additional Internet usages [5]. 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 codes. This improvement brings two threats[6, 11]:

• Infinite loops

• Public accessibility

In this section, we discussed the latter. A blockchain user has the entire chain on his 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[11, 5], every participant in Ethereum receives a virtual machine that separates her 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 Smart Contract?

In a nutshell, smart contracts are merely contracts. The main differences between these contracts and the common "real world" contracts are [6, 10, 11, 13, 8]:

- 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 [13], where the writer uses the idea carved on stone contracts. These contracts provide a high level of trust. In a more modern example [13], we can think of 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 fig 1 that this procedure can be programmed [13]. We can create more sophisticated contracts

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

Figure 1: Vending Machine Contract

[7] as one can see in figure 2

2.3.1 Smart Contract Compilation

We use a special language program for smart contracts: **Solidity**, [14, 11]. An example of a Solidity code [9] can be seen in 3, and the entire compilation process[9] in 4 We can see that there are two outcomes for the output of the compilation process :

- Application Binary Interface **ABI** This interface clarifies which functions and which parameters the contract uses [14, 9, 17]
- Bytecode A hexadecimal string that encodes the source code. The developer uploads the bytecode to the EVM. The EVM extracts the opcode instructions [14, 9]. An example opcode can be seen [9] in 5 respectively.



Figure 2: Selling car using a smart contract

```
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

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



Figure 4: Smart contract's compilation chain

Ethereum handles this obstacle [6, 14, 11]. Every source code is an ordered collection of commands. Thus, Ethereum uses the concept of **Gas** [11]. The idea is that one has to pay for every computation he wishes to run 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

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 .crypto-currency.

3 Machine Learning

Machine learning **ML** is not a new discipline. Researchers used tools such as decision trees and probabilistic modeling for decades[21, 22]. The usage of ML has been boosted

[00]	PUSH1	80
[02]	PUSH1	40
[04]	MSTORE	
[05]	CALLVALU	E
[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

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 [21, 22, 25] and XGBoost [23, 24, 21] are known to most researchers, in this section I will describe some of the deep learning tools that we used.

3.1 Generative Pre-trained Transformer-GPT

The non-quantitative nature of text challenged the DL world from its inception. Researchers searched for an optimal method to represent text numerically in a way that preserves its intrinsic semantic properties [26, 27]. In 2017, researchers achieved a breakthrough when they presented a novel architecture: the **transformers** [28]. Describing transformers in detail is beyond the scope of this paper. However, one can learn its general structure in 6. The left part of the transformer is called "encoder" and the right "decoder" The DL world has focused on transformers since they pre-



Figure 6: Transformers' Architecture

sented for the first time.In 2018, **OpenAI** published a new type of transformer: **GPT** [29]. This architecture is comprised only of the "decoder" part. Hence, it is easier to be trained 7. As you may assume, this architecture has advanced versions **GPT2** and **GPT3.5**. We will discuss the former in the next section, and the latter is the platform for the first version of ChatGPT,

3.2 Auto-Encoder

Auto-encoder [21] was one of the first neural network architectures. Most of the neural network architectures focus on tasks such as prediction. Auto-encoder maps the real data into an Euclidean representation as in 8. The auto-encoder consists of two parts:

- **Encoder** Encoder maps the data to a latent space (usually with a lower dimension).
- Decoder- Decoder maps the latent space back to the raw space

When we complete the training process, the encoding component receives real data as input and outputs an Euclidean representation in a lower dimension.



Figure 7: GPT-Decoder Only

3.3 Bayesian Network

Causal inference is a rising research domain [31, 32]. It offers promising results in various disciplines of applied science. Some of its tools can enhance the methodologies



Figure 8: Auto Encoders [30]

by which researchers approach data [31, 32]. A core object in Casual inference is Directed Acyclic Graph **DAG**, [31, 32]. It represents the actual relations between variables and allows us to perform casual analysis 9.



Figure 9: Directed Acyclic Graph - AG

However, in real-world data, the arcs' directions in a data graph are often unknown. Researchers use probabilistic tools to approximate these directions. Bayesian networks are such tool [33, 34, 35]. The idea is that statistical tools will denoise the data and reveal the genuine mutual influence between given variables, as one can see in fig 10. In the following sections of this paper, we will describe Ethereum ML research that



Figure 10: Bayesian Network [36]

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

4 Reentrancy Attack-Real World Example

We saw in Smart Contracts 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 its 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 [12, 18, 19, 20, 14]. One of the most common attacks of smart contracts is the **reentrancy attack**. In such an attack, a fund transaction takes place. Rather than updating the paying account and transferring the fund in the following step, the contract transfers the fund 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 fig 11. This scheme is presented in [18, 19]



Figure 11: Reentrancy Scheme

We present a Solidity implementation of the reentrancy attack in figures 12 and 13 [18] The most famous **reentrancy attack** is the **DAO Attack**. It took place in



Figure 12: Deposit Function

2017. 60 million dollars were stolen. Moreover, the main outcome of this attack was a hard fork of Ethereum, that led to two communities. The readers can learn about this attack and its impacts on Ethereum [6, 11, 12]. There were some additional attacks after the DAO such as [18]:



Figure 13: Reentrancy Attack function

- Uniswap/Lendf.Me hacks (April 2020) 25 Million dollars, attacked by a hacker using reentrancy.
- The BurgerSwap hack (May 2021) 7.2 Million dollars because of a and a reentrancy exploit.

5 Research Methodology

We will present three ML types of research that focus on smart contracts:

- Smart contract detection based on the opcode. This model relies on the GPT2 model [29, 38, 39]
- Smart contract detection based on the opcode. This model relies on CodeLlama fine-tuning [43, 44, 45]
- Malicious transaction detection which is based mainly on gas-driven variables.

In these studies, we develop ML binary models using common tools Machine Learning or [29, 38, 43, 45]. In ML Study of Malicious Transactions, we describe in detail the transactions model. We assume that two main manners characterize malicious transactions:

- 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

These two assumptions gave us the guidelines for choosing the relevant variables to our model. In the smart contract models, we followed the code generation approach [29, 45] and used the code LLM models. For the solidity study, we "innovatively" fine-tuned a CodeLlama binary model [46].

5.1 Malicious Data

When we develop binary models we need labeled data for both populations (e.g. benign and malicious). While benign data is always available (as an example see [37]), there are some obstacles in aggregating malicious data:

- Both Solidity sources and ABI of malicious contracts are seldom available
- It is difficult to label contact as malicious unless it is already harmful.
- Transactions are rarely labeled as malicious

We, therefore, used common websites such as [41, 52] that indicate exploiting contracts or scam transactions. It allowed us to collect "bad reputation" contracts and indicate them as malicious

6 The Opcode project

6.1 Smart Contracts' Detection

The idea of using an opcode source for detecting malicious smart contracts is not novel [37]. Opcode is expected to be a beneficial prediction feature due to several reasons:

- The code is easy to disassemble from the bytecode
- The commands are human-readable, which allows NLP tools usage.
- We can detect statistical patterns on verified and non-verified contracts [37].

We aspire to develop a classification model that receives opcode sources of smart contracts and detects whether they are malicious. We decided to use GPT2 [38] that is provided by Huggingface [39]. The motivation to use this model is that it simply allows us to consider each contract source as a textual section where each command is a word. Our data included 10k benign smart contracts and about 800 malicious ones. Namely, our data is extremely imbalanced. Before starting the training, we measure the performances of GPT2 as presented in 14 One can see that GPT2 performs nearly



Figure 14: GPT2 False Positive Vs. recall

as random. We use GPT2's weights as our initial weights and train its architecture using the data that we have described. In early trials, our naive approach was flattening each contract and providing it to the model. The results weren't good, which led us to perform some trial and error for improvements. Truncating the prefix and restricting the length to 600 commands (with padding if needed) provided optimal results, as shown in fig 15.



Figure 15: Trained GPT2 False Positive Vs. recall

6.2 Opcode Graphs

As in nearly every other ML project in cyber, prediction is insufficient: one needs to demystify which features are cardinal in determining the model's decision. In our project, there are no features besides the opcode commands. Thus, we can focus only on studying the command occurrences and their inner relations, such as mutual occurrence (or anti-occurrence). We used Python's bnlearn [34] to generate the DAGs of the benign and the malicious contracts. We present the comparison between the two DAGs in figure 16

The main objective of this comparison is to show that a difference between the two DAGs, particularly their arcs, exists.

7 The Solidity Project

The success of the Opcode project motivated us to develop another prediction model: a Solidity model. After a short research, we realized that:

- There are no available databases of malicious contracts.
- A malicious contract is often unverified -we don't have the source.

The latter requires a remark about Ethereum manner: In contrast to bytecode, we don't upload Solidity sources to the blockchain. When we deploy a contract, we don't have to verify it, which leads the source to remain unobservable. This manner is counter "blockchain": we would expect that in an environment that aspires to be social, the community has to verify a code and not the developer. We needed both to increase our malicious data compared to the opcode project and generate source code. We achieved the former using some websites such as Chainabuse [41] and the latter using a web decompiler [40]. 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 [42].



(b) Benign DAG

Figure 16: DAGs Comparison

7.1 CodeLlama

We wish to use the data to train a model. Since we are in the large language models **LLM** era, a natural step is searching for such a model. Llama is a collection of text generation LLMs that Meta trained [43, 44], and includes code generative models **CodeLlama**. For training these models, Meta used Java and Python sources [45] for training these models, which can fit our needs. We need to fit them for classification tasks, which requires the following:

- Modifying the architecture
- Perform a model fine-tuning

The second clause is mandatory since the training process of the basic model included different programming languages. We wish to focus solely on Solidity and its maliciousbenign differences. The solution of the first clause is described in detail in [46]. We managed to achieve results of around 70 percent accuracy. However, it is a 7B model with very little data. Since we cannot increase our data massively (we don't have sufficient malicious contracts), we searched for a smaller model.

7.2 The Solidity Generator

While making a quick search on the web, we found a solidity generator model [47], which is based on **BERT** [48] architecture. We used a similar fashion to [46] to modify it. However, this model is significantly smaller (about 500M). Thus, we can better both validate its results and deploy it. We present the results in fig 17 Before we analyze



Figure 17: The Solidity Classification Model

the results, we describe the axes KPI's

- X-axes: Precision- The probability that a detected contract is malicious.
- Y-axes: Recall The probability of detecting a malicious contract.

The base model is the model before we fine-tuned it. On one hand, the results are around 80 percent, which is an outcome of the absence of data. On the other hand, our fine-tuned model is significantly better than the base model. This indicates that the data is "trainable". We have a good reason to assume that enriching the data, particularly with malicious sources, will provide a better model.

8 ML Study of Malicious Transactions

In this section, we discuss the ML study of the transactions. We recall that we restrict ourselves to transactions in which the receiver is a smart contract. We will base our model features on data, that we get from the web3 package [49].

8.1 Data

We must aggregate malicious and benign transactions to train a malicious detection engine. For aggregating malicious transactions we used websites such as **de-fi** and **chainabuse** [51, 52] as well as information about malicious smart contracts that one can easily find [51]. We aggregated about 15K malicious transactions. For aggregating benign transactions we used two methods:

- Sampled "modern" transactions (namely from the very last blocks.
- Sampled from blocks we took malicious transactions to overcome potential data temporality.

For generating features we used the structures of **gettransaction** and **gettransactionreceipt** of Python's web3 [49].

8.1.1 Our Features

The web3 package provides a massive amount of data about the Ethereum transactions. Not all of them provide information that is beneficial for malicious detection. Our research focused on variables that describe aspects of gas bid-ask, such as the value, and (as we present in the next subsection) the input's 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 reflects the gs pricing, in particular, it reflects if the transaction follows EIP-1559 or not.
- 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

We can generate ML features from these variables. These features are the input of an ML model. The motivation to focus on gas consumption, gas price, and gas fee, is derived from the assumption that a potential culprit will offer a high amount of gas to execute his transactions.

8.2 The Model

Since we are using tabular data, the natural approach is using XGBOOST [23, 24] The input features we provided XGBOOST are the variables we discussed in the previous section with some additional aggregators. We trained the XGBOOST model and got the following ROC: In addition, we present the feature importance of the xgboost model: We can see that the gas variables are the most significant as one may expect.



Figure 18: ROC curve of the xgboost model



Figure 19: 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 their actual smart contracts' functions. Our next step in the research is using this map to generate statistics about such functions. We developed a mapping from the Hexa signatures to the functions. We denote the latter **hexdic**. The following graph represents the distribution of signatures: One can see that most of the signatures that we can map appear only in benign transactions. on the other hand, we have about 60k benign transitions and about 100 signatures appear only in the malicious which is a potential indicator. The next KPI that we studied was the length of each input. The results are presented One can see differences between the distribution lengths which we may use for detection using a Bayesian or a frequentist fashion. The following graph represents the distribution of the valid amount of signatures (valid -we have in our map)

To provide some expert knowledge, we present a list of functions, that are common for both being and malicious Finally, we wish to add this information to the model. The following source presented one of our approaches

def generate_octet_features(self,tx_input):



Figure 20: Signatures' histogram



Figure 21: Signatures' Length Distribution

```
list_of_ordered_octets = textwrap.fill(tx_input[2:], 8).split()
n_octets =len(list_of_ordered_octets)
valid_octet=0
mal_octet=0
benign_octet=0
for octet in list_of_ordered_octets:
   mm = '0x' + octet
    if '0x' + octet in self.hex_dic:
        valid_octet += 1
        if self.hex_dic[mm] in self.mal_dic:
           mal_octet += 1
           continue
        if self.hex_dic[mm] in self.benign_dic:
            benign_octet += 1
            continue
return n_octets, valid_octet, benign_octet, mal_octet
```

The model results are presented in the following ROC: We add the feature importance as well: It can be seen that the results are slightly better but not in a significant way. We can explain it by the fact that many transactions are empty. We can deduce that the 4bytes method can be beneficial as an anomaly tool that assists xgboost.

8.4 DL Early Study

in many scenarios, it is difficult to have malicious transitions. We train an auto-encoder only with benign transactions to detect anomalies. We projected the representation, on a PCA space [50], as shown in figure 26 :



(b) Prop valid

Figure 22: Valid signature Distribution



Figure 23: Valid signature Distribution



Figure 24: Model's ROC with 4bytes



Figure 25: Model's Features' importance with 4bytes

One can see that we achieve a separation between the malicious transactions and the benign ones. The graph doesn't indicate how well the separation is, but it suggests that DL methods can be beneficial in detecting differences between the two transaction populations.

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



(a) Components 0 1 2



(b) Components 2 3 4



(c) Components 0 1 4

Figure 26: Transactions Projected on PCA

References

- [1] ChatGPT https://chat.openai.com/
- [2] Dall-E https://openai.com/dall-e-2
- [3] Sentiment Analysis, https://en.wikipedia.org/wiki/Sentiment_analysis
- [4] Etherscan https://etherscan.io/
- [5] A. Moskov, (2017). What is Ethereum The ultimate beginner's guide, https://coincentral.com/what-is-ethereum-the-ultimate-beginners-guide
- [6] V. Buterin, (2014). DAOs, DACs, DAs and More: An Incomplete Terminology Guide. https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-anincomplete-terminology-guide
- [7] Smart contract platforms: overview https://coinloan.io/blog/smart-contractplatforms-overview
- [8] INTRODUCTION TO SMART CONTRACTS https://ethereum.org/se/developers/docs/smart-contracts/
- [9] Zaryab Afser EVM Part ——: The Journey of Smart Contracts from Solidity code to Bytecode https://medium.com/coinmonks/evm-part-ii-the-journey-ofsmart-contracts-from-solidity-code-to-bytecode
- [10] Cryptorunner https://cryptorunner.com/what-is-ethereum/
- [11] Udemy-Blockchain A-Z https://www.udemy.com/course/build-your-blockchainaz
- [12] M. Leising, (2017), The Ether Thief https://www.bloomberg.com/features/2017the-ether-thief
- [13] N. Custodio, (2017), Smart Contract for Dummies, https://www.freecodecamp.org/news/smart-contracts-for-dummies-a1ba1e0b9575/
- [14] Vanunu, Zaikin, Barda (2023), Cyber and Hacking in the Worlds of Blockchain and Crypto
- [15] https://ethereum.org/en/what-is-ethereum/
- [16] N. Custodio, (2017), https://www.freecodecamp.org/news/smart-contracts-fordummies
- [17] Louis Abraham's Home Page, (2022). Calling a contract without ABI on Ethereum https://louisabraham.github.io/articles/no-abi
- [18] @kamipolak,(2022), Hack Solidity: Reentrancy Attack https://hackernoon.com/hack-solidity-reentrancy-attack
- [19] Reentrancy Attack in a Solidity Smart Contract https://cryptomarketpool.com/reentrancy-attack-in-a-solidity-smart-contract/
- [20] The Ultimate Guide To Reentrancy https://medium.com/immunefi/the-ultimateguide-to-reentrancy-19526f105ac
- [21] Stnadford Course, https://cs229.stanford.edu/
- [22] R. O. Duda, P. Hart, D. G. Stork Pattern Classification. Wiley, (2000)
- [23] xgboost 2022, https://xgboost.readthedocs.io/en/stable/,
- [24] https://xgboost.readthedocs.io/en/stable/tutorials/model.html

- [25] T. Yiu, Understanding Random Forest https://towardsdatascience.com/understandingrandom-forest
- [26] T. Mikolov, M. Karafiat, et.al InterSpeech2010, Recurrent neural network based language model
- [27] I. Sutskever, O. Vinyals, Q. V. Le, (2014). Sequence to Sequence Learning with Neural Networks https://arxiv.org/abs/1409.3215
- [28] A. Vaswani, et al, (2017). Attention Is All You Need https://arxiv.org/abs/1706.03762
- [29] A.Radford, K.Narasimhan, T. Salimans, I. Sutskever, 2018. Improving Language Understanding by Generative Pre-Training
- [30] https://www.compthree.com/blog/autoencoder/
- [31] A. Molak, (2023) Packt Publishing, Causal Inference and Discovery in Python: Unlock the secrets of modern causal machine learning with DoWhy, EconML, Py-Torch and more
- [32] B. Neal, Causal Inference. https://www.youtube.com/c/BradyNealCausalInference
- [33] bnlearn https://www.bnlearn.com/examples/
- [34] Pybnlearn package https://pypi.org/project/bnlearn/
- [35] N Katz Explainability Using Bayesian Networks https://medium.com/towardsdata-science/explainability-using-bayesian-networks-4dc706680294
- [36] University of Bergen https://www.uib.no/en/rg/ml/119695/bayesian-networks
- [37] How Forta's Predictive ML Models Detect Attacks Before Exploitation, (2022).https://forta.org/blog/how-fortas-predictive-ml-models-detect-attacksbefore-exploitation/
- [38] A.Radford, J. Wu, R. Child, D. Luan1, D. Amodei, I. Sutskever, (2019), Language Models are Unsupervised Multitask Learners
- [39] https://huggingface.co
- [40] Ethereum decompiler https://ethervm.io/decompile
- [41] Chainbuse https://www.chainabuse.com/
- [42] https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts
- [43] https://ai.meta.com/llama/
- [44] Huggingface Llama, huggingface.co/docs/transformers/model_doc/llama2
- [45] CodeLlama https://huggingface.co/docs/transformers/main/model_doc/code_llama
- [46] N Katz, CodeLlama FineTuning for Classification, https://medium.com/@natankatz/codellama-classification-finetuning-28fa5546f64f
- [47] Solidity Generator, https://huggingface.co/ckandemir/solidity-generator
- [48] BERT Paper https://arxiv.org/abs/1810.04805
- [49] Web3 Package https://web3py.readthedocs.io/en/stable/
- [50] Z. Jaadi A Step-by-Step Explanation of Principal Component Analysis (PCA) https://builtin.com/data-science/step-step-explanation-principal-componentanalysis
- [51] https://www.chainabuse.com/
- [52] https://de.fi/
- [53] https://www.4byte.directory