A Survey on Ethereum Smart Contract Vulnerability Detection Using Machine Learning

Onur Sürücü*[a], Uygar Yeprem[c], Connor Wilkinson[b], Waleed Hilal[b], S. Andrew Gadsden[b], John Yawney[a], Naseem Alsadi[b], Alessandro Giuliano[b]

[a]Adastra Corporation, 200 Bay St. Suite, Toronto, ON, Canada; [b]University of McMaster, 1280 Main St. W, Hamilton, Hamilton, ON Canada; [c]University of Guelph, 50 Stone Rd E, Guelph, ON Canada

ABSTRACT

Blockchain applications go far beyond cryptocurrency. As an essential blockchain tool, smart contracts are executable programs that establish an agreement between two parties. Millions of dollars of transactions attract hackers at a hastened pace, and cyber-attacks have caused large economic losses in the past. Due to this, the industry is seeking robust and effective methods to detect vulnerabilities in smart contracts to ultimately provide a remedy. The industry has been utilizing static analysis tools to reveal security gaps, which requires an understanding and insight over all possible execution paths to identify known contract vulnerabilities. Yet, the computational complexity increases as the path gets deeper. Recently, researchers have been proposing ML-driven intelligent techniques aiming to improve the efficiency and detection rate. Such solutions can provide quicker and more robust detection options than the traditionally used static analysis tools. As of this publication date, there is currently no published survey paper on smart contract vulnerability detection mechanisms using ML models. In order to set the ground for further development of ML-driven solutions, in this survey paper, we extensively reviewed and summarized a wide variety of ML-driven intelligent detection mechanism from the following databases: Google Scholar, Engineering Village, Springer, Web of Science, Academic Search Premier, and Scholars Portal Journal. In conclusion, we provided our insights on common traits, limitations and advancement of ML-driven solutions proposed for this field.

Keywords: smart contract, Ethereum, blockchain, security, vulnerability detection, artificial intelligence, machine learning

1. BRIEF INTRODUCTION

A blockchain is a digitally distributed and shared transaction ledger, shared amongst the nodes of a computer network (miners). All nodes are governed by a consensus protocol, which refers of transaction rules and states to achieve agreement and security across computer network [1]. By this inherent design, the information on blockchain (i.e., transaction data) are immutable and indelible.

Smart contracts are computer programs or transaction protocols stored on blockchain that execute when predetermined conditions are met. These contracts are just a collection of codes and data (state), which resides at a specific address on the Ethereum blockchain. They are commonly being used to automate an execution of an agreement without needing a third party (i.e., bank or government). For instance, contracts used for financial applications are typically for managing, gathering or distributing an asset. Additionally, its immutability feature makes it a perfect option to store important data (i.e., ownership, provenance) for notary purposes. The majority of smart contracts run on Ethereum blockchain (market capitalization exceeding $400 billions), and they are being powered by a programming language known "Solidity" [3]. The scripts typically contain simple conditional statements ("if/when...then…") for managing the given assets, similar to a paper contract. Each piece of code in the smart contract is executed sequentially and no parties can influence the code execution. When predetermined conditions are met, the execution process is done across a network of computers. Nowadays, decentralized applications (DApps) operate autonomously, by providing a user interface as frontend and utilize smart contracts as the backend. DApps promises more transparency compared to the conventional applications. For example, intentional cheating in local places in an organization cannot happen in a transparent ecosystem, since the ledgers are publicly accessible. However, since these contracts can manage billions of dollars of virtual assets, they become an attractive target for hackers. In June 2016, malicious individuals attacked decentralized application organization's (DAO) contracts by utilizing re-entrancy vulnerability to steal 3.6 million Ether ($10 billion US Dollars)

Disruptive Technologies in Information Sciences VI, edited by Misty Blowers, Russell D. Hall, Venkateswara R. Dasari, Proc. of SPIE Vol. 12117, 121170C · © 2022 SPIE · 0277-786X · doi: 10.1117/12.2618899

[4]. Thus, a simple developer negligence in smart contracts can cause the loss of millions of dollars. Due to that reason, development for effective vulnerability detection tools has been ongoing, as the utility of DAOs are increasing. The conventional statistical analysis tools for detecting weaknesses in smart contracts purely rely on manually defined patterns, which are likely to be error-prone and can cause them to fail in complex situations. As a result, expert attackers can easily exploit these manual checking patterns. To minimize the risk of the attackers, machine learning powered systems provide more secure solutions relative to hard-coded static checking tools.

Machine learning (ML) is a branch of artificial intelligence, which uses algorithms to automatically learn by observing prior data, and it has a capability of improving itself, similar to a human being. ML models are considered as black boxes since the end-user does not necessarily fully know how the model makes its decision. Therefore, ML technology is a feasible solution for detection of vulnerabilities in smart contracts. Many researches have inclined towards ML-driven solutions for security issues [12]-[27].

Throughout our research, we noticed the future direction of the literature on smart contract vulnerability detection, and our goal is to provide guidance for new developments in this field. In the academia of smart contracts, there is no published survey paper on ML-driven smart contract vulnerability detection models, as of the date of this paper's publication. In order to set the ground for further development of ML method on smart contract vulnerability detection, in this survey paper, we reviewed many ML-driven intelligent detection mechanism on the following databases: Google Scholar, Engineering Village, Springer, Web of Science, Academic Search Premier, and Scholars Portal Journal. We provided our insights on limitations and advancement of ML-driven solutions.

2. SECURITY ISSUES OF SMART CONTRACTS

In this section, we introduce common key vulnerabilities in smart contracts that can be exploited by malicious individuals. In order to provide a finer demonstration of vulnerabilities, a further information on blockchain technology and smart contract is provided in the initial portion of this section (section 2.1). Later, we described common vulnerabilities that ML models addressed (section 2.2).

2.1 Overview of Smart Contract Technology

A blockchain is a chain of blocks (records), where the blocks are linked (chained) and secured utilizing cryptography. A simple analogy of blockchain is a ledger, where blocks are similar to pages holding records. Each block contains the following records: a transaction data, a time stamp, and the hash value of the previous block (parent block) and a nonce, which is a random number for verifying the hash [5]. The blockchain is stored on a network of nodes (computers), where all nodes have the copy of the blockchain. In other words, all nodes (*miners*) is required to store blockchain data on their local system while synchronizing all of their block with those stored by other nodes based on a consensus model [6]. Due to that reason, everything inside the blockchain is publicly visible, in other words, transparent. Once a block is on the blockchain, it can't be changed unless all nodes are agreed to (Consensus mechanism). The most popular public blockchain platform for smart contracts is *Ethereum*. Also, Ethereum has its own currency known as *Ether* and it can be transferred between accounts same as the other currencies. Ethereum network is aimed to provide a decentralized Turing-complete machine (Ethereum Virtual Machine) by executing scripts using public nodes located in all around the world. Through Ethereum, one can use programming languages (e.g., *Solidity*) to build smart contracts. Before deploying the contract, the contract's written language is converted to *Ethereum bytecode*, where the converted bytecode is deployed to the Ethereum blockchain. In order to deploy the contract to the blockchain, the miners must be compensated for the computing energy required to validate the transaction and execute the smart contracts code, where the fee is named as *gas fee* [3].

Due to nature of blockchain (i.e., immutability), smart contracts are relatively more susceptible to vulnerabilities than other digital systems [7]. First, the second a contract is on a blockchain, any other contracts or individuals can invoke it. Thus, the input combination can be anything, and there will be always an untested execution path of the deployed contract. Therefore, in testing, covering as many of input combination as possible is critical. Secondly, if a deployed contract contains a vulnerability, the developer cannot update the contract, since it is stored in an immutable ledger of blockchain.

2.2 Smart Contract Vulnerabilities

Re-entrancy vulnerability is occurred when a function invokes an untrusted contract [8]. Some smart contracts need to have interaction between external contracts to complete the transaction. In that case, a user can invoke a deployed smart contract by utilizing deployed contract's unique address. All smart contracts have an unnamed function known as a fallback function, in which no argument nor return value exist. In solidity, the call function transfers Ethers by invoking a method

in source code or a method in an external contract. For instance, if a call method is being used to transfer an asset to sender's account, it will automatically call invoke sender's fallback function. This process does not have any limitation for call method invocation and the fallback function can be executed until allocated gas amount is consumed. Figure 1 shows a code snippet written in Solidity, which contains two simple smart contracts: victim and attacker [8]. Victim contract acts like a bank, where it has a withdraw function for transferring Ether to the caller (line 4). At the end of the transaction, the caller's fallback function is invoked (line 10). The attacker utilizes the following scenario to steal Ether from the callee: (1) Attacker starts the transaction by calling victim's withdraw function; (2) victim executes the transaction (line 4) and invokes the fallback function (line 10-12); (3) The fallback function recursively calls the withdraw function again and again (re-entrancy); (4) Until the exit condition is satisfied, victim's withdraw function sends ether to the attacker.

```
1   contract Victim {
2     bool flag = false;
3     function withdraw() {
4       if (flag || !msg.sender.call.value(1 wei)()) throw;
5       flag = true;
6     }
7   }
8   contract Attacker {
9     uint count = 0;
10    function() payable {
11      if(++count < 10) Victim(msg.sender).withdraw();
12    }
13  }
```

Figure 1. A code snippet to demonstrate re-entrancy vulnerability [8].

In some cases, the developer sets condition based on the block timestamp to execute some critical operations. When, a miner mines a new block, a timestep must be provided along the other information, and the block timestamp is linked to the miner's local computer or server's clock. Due to the blockchain's nature, the timestamp of a block can vary up to 900 seconds with other miner's timestamps [9]. Therefore, if a smart contract utilizes "now" to invoke a critical method, then a malicious miner can manipulate the code by alternating the timestamp, namely *timestamp dependence vulnerability*.

An *infinite loop vulnerability* is a common logic error in all programming languages. These errors usually occur in a function with looping statements (for, while, or self-invocation loop) without a proper exit condition, known as infinite loop. In this case, the deployed smart contract would run until it runs out its gas, without fully completing its functionality.

A block on a blockchain consists of past transactions, and the blockchain state is updated numerous times during each period. There is no guarantee that the given transaction will be completed in sequence. So, the actual state of the smart contract is unpredictable. Therefore, if two independent transactions are executed to invoke the same smart contract, which then the order of the execution is decided by the miners. If the attacker is the miner, then the transactions can be rearranged in such a way that the result would benefit him/her, namely *transaction ordering dependency vulnerability* [10].

An *integer overflow/underflow vulnerability* can occur, when a variable's value or size exceeds its upper or lower limits during a computation. For instance, if an account's balance is at its lower or upper limit, then the variable's value is reset to zero. In the past, an anonymous hacker drained off 2000 Ethereum (is worth more than $2.3 million) by manipulating this vulnerability [11].

3. MACHINE LEARNING METHODS FOR VULNERABILITY DETECTION

3.1 Feature Engineering for Smart Contracts

Feature engineering is the process of selecting, extracting, and transforming raw data into features that contains more information, so that the ML model can recognize given input vector's pattern. Due to that reason, various feature extraction methods are being used with robust and viable ML methods. For smart contracts, various feature engineering techniques

are being used to represent internal dynamics of the source code. Here, we briefly explained common feature engineering techniques that has been used along with intelligent ML methods to increase the visibility of essential patterns.

Abstract syntax tree (AST) is a data structure that being used to reason a written language. The source code is converted using compilers into a data structure tree, where each node represents a syntactical element of the source code, and the tree shaped diagram displays the flow of each element inside the written program. Without disrupting the structured information, this method can provide details of the source code (e.g., number of functions) [17]. By analyzing a source code's AST, a ML model can identify common traits of a secure or weak contract.

Control flow graph (CFG) is a graph notation for representing all possible execution paths a program can handle. Usually, a smart contract's opcode is utilized to analyze the execution paths of the program. The CFG method is mainly used for compile optimization and static analysis tools. Inside the graph, each node represents a basic block, where a jump in the source code starts a block, with the same principle, a jump target ends a block [18]. Thus, inside a block no jumps are occurred. As a last element in the graph, directed edges represent the action of jumping in the source code.

Opcode (operation codes) is a human readable representation of bytecodes. It is the portion of a machine language instruction that contains a list of tasks or operations to be executed by the computer [19].

3.2 Existing Analysis Tools

As blockchain and digital currency become ever-prevalent internationally, detecting vulnerabilities in smart contracts has become an important problem. Attacks such as the DAO bug [12] or the freezing bug in the Parity multisig (multisignature) wallet [13] cost blockchain users hundreds of millions of dollars. As a result, several security tools aimed to identify and prevent such vulnerabilities. One of the earliest methods created was called Oyente [14], which successfully identified vulnerabilities such as the DAO bug vulnerability across roughly smart 9,000 of 19,000 contracts. It uses symbolic execution on EVM bytecode to detect issues within smart contracts. Due to its early implementation, and static ruleset, many newer vulnerabilities are not detected by Oyente, and as a result, has been improved upon by countless other methods. Most conventional security tools are similar to Oyente, in that their ruleset is predefined and as a result, they will not automatically adjust when new vulnerabilities are introduced. A similar but improved version of Oyente evolved a few years later, when SmartCheck was introduced by E. Marchenko [15]. SmartCheck focuses solely on vulnerabilities for smart contracts written in Solidity, Ethereum's base programming language. It runs analysis on the syntax and linguistics used within the smart contracts, and detects vulnerabilities based on XPath patterns [16]. Unfortunately, its reliability on predefined XPath queries pose a similar issue to Oyente, wherein the defined XPath patterns decide the accuracy and effectiveness of SmartCheck. Additionally, they leave little room for growth and dynamic analysis, since the defined patterns are static. As a result, many researchers are attempting to utilize the dynamic capabilities of machine learning to construct novel vulnerability detection techniques, that outperform such traditional methods. This paper aims to review these newly developed approaches.

3.3 Deep Learning Models

Deep learning is a subfield of machine learning, where the algorithms are inspired by neural networks (NN). These neural nets essentially attempt to mimic the working principle of a human brain. The main difference with a classical machine learning algorithm is the data type (unstructured data) that it requires, and the learning strategy that it uses.

In 2018, Goswami et al. mentioned that while existing symbolic tools (e.g., Oyente) for analyzing vulnerabilities have proven to be efficient, their execution time increases significantly with depth of invocations in a smart contract [20]. They proposed an LSTM neural network model to detect vulnerabilities in ERC-20 smart contracts in an effort to produce a less time consuming and efficient alternative to symbolic analysis tools. The preprocessing steps followed in this paper were very similar to the methods used by [21]. The model was trained and tested on a dataset of 165,652 ERC-20 smart contracts, which consisted of bytecode data labeled by Maian and Mythril (statistical code analysis tools). The proposed model achieved 93.26% accuracy, 92% recall and an F_1 score of 93% on the testing set. Further they have compared the time performance of their model to those of the symbolic analysis tools Maian and Mythril (static analysis tools). While their proposed model had a runtime of 15 seconds on a testing set of 5,000 random tokens, Maian and Mythril took 32,476 and 9,475 seconds respectively. These results indicate the same type of improvement achieved over symbolic analysis tools as in [20].

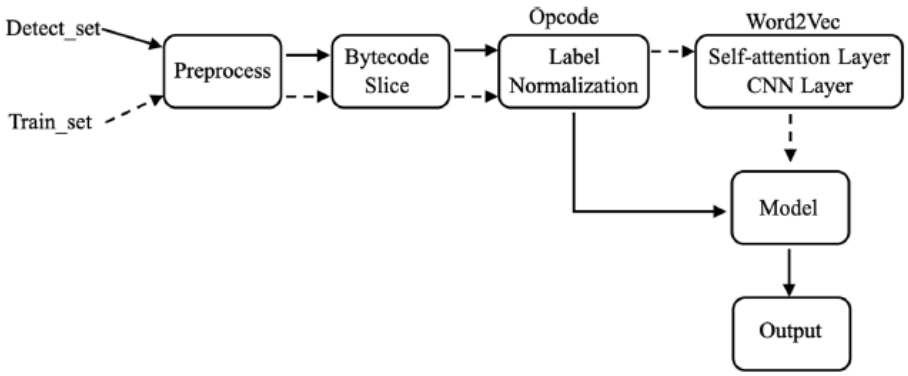

Figure 2. Model with Self-Attention + CNN Overview [20].

In 2018, Liao et al. have adopted a sequence learning approach to detect smart contract security threats [21]. Smart contract data was obtained from the Google Big Query Ethereum blockchain dataset. Ultimately, an LSTM model was trained on 620,000 contracts from this source. Once again, the derived opcodes from the contracts were represented as one-hot vectors. As this type of representation results in highly sparse and uninformative features, these vectors were transformed into code vectors using embedding algorithms, resulting in lower dimensionality and a higher capability of capturing potential relationship between sequences. As another preprocessing step, they have compared the statistical properties of the opcode lengths of contracts that were identified as vulnerable and safe. Having observed that the properties of the two categories differ significantly, they have limited the input data to the LSTM to only include contracts that had a maximum opcode length of 1600, as a design choice. Further, the distribution of the dataset (labeled by MAIAN) was realized to be imbalanced with non-vulnerable instances making up 99.03% of the dataset. Therefore, all vulnerable contracts were grouped together and oversampled to achieve a balanced distribution in the training set using the Synthetic Minority Oversampling Technique (SMOTE). The results indicated the superiority of a sequential learning approach over symbolic analysis tools. The model achieved a vulnerability detection accuracy of 99.57% and F_1 score of 86.04%.

In 2019, SoliAudit model was proposed to enhance the vulnerability detection of smart contracts [22]. Smart contract source code in Solidity is converted into an opcode sequence to preserve the structure of executions. Each contract goes through both a dynamic fuzzer and a vulnerability analyzer. The vulnerability analyzer consists of a static machine learning classifier, which detects vulnerable classes, whereas the fuzzer (this term was introduced in an earlier paper) will parse the Application Binary Interface (ABI) of a smart contract to extract its declared function descriptions, data types of their arguments and their signatures. It will then return the smart contract inputs and functions that are identified as vulnerable. The idea of a smart contract fuzzer was introduced by the authors of [22]. Vulnerability analyzer used a set of labels (13 vulnerabilities) determined by analysis tools such as Oyente and Remix. Before training the opcode sequence data using these labels, two types of feature extraction methods were tested. These were namely, n-gram with tf-idf and word2vec. The experiments were carried out by applying the former method together with algorithms such as Logistic Regression, Support Vector Machine, K-Nearest Neighbor, Decision Trees, Random Forests and Gradient Boosting. The output from the latter (word2vec) was a matrix and a Convolutional Neural Network (CNN) was preferred to train it as it considers the inner structure of the matrix. However, this combination of feature extraction and training did not yield good results. The best results for the classification of vulnerabilities were obtained using Logistic Regression with an accuracy of 97.3% and F_1 score of 90.4%.

In 2019, similar to N. Lesimple, the authors in [23] explored the use of LSTM in the context of detecting smart contract vulnerabilities. Specifically using Average Stochastic Gradient Descent Weight-Dropped LSTM (or AWD-LSTM), A. Gogineni et al. attempted to showcase the reduced search time and increased accuracy of such a model, when compared to traditional models. They focus on four known vulnerabilities and showcase a weighted average F_{beta} score of 90% when compared against labels generated from traditional techniques. Similar to N. Lesimple's results, the study is hindered by it's dependency on gathering labels from traditional methods, not allowing it measure it's relative accuracy to such methods. Though, this paper does identify that ML techniques can allow vulnerability detection techniques to produce accurate results in a more efficient fashion, introducing a more scalable approach.

In 2020, Xing et al. [24] developed a new feature extraction method called slicing matrix, which consists of segmenting the opcode sequences derived from smart contract bytecodes to extract opcode features from each one

individually. The purpose of this segmentation is to separate useful and useless opcodes. The extracted opcode features are then combined to form the slice matrix. To carry out a comparative analysis, three models were created. These were namely Neural Network Based on opcode Feature (NNBOOF), Convolution Neural Network Based on Slice Matrix (CNNBOSM), Random Forest Based on opcode Feature (RFBOOF) [24]. These three models were each tested on three different vulnerability classification tasks: greedy contract vulnerability, arithmetic overflow/underflow vulnerability and short address vulnerability. While RFBOOF achieved the best results in all three cases based on precision, recall and F_1 evaluation metrics, CNNBOSM performed slightly better than NNBOOF in general. The authors mention that the slice matrix feature need further exploring.

In 2020, In N. Lesimple et al.'s paper [12], the authors study the effect of deep learning models when used to identify vulnerabilities in Smart Contracts. It specifically highlights the vulnerabilities relating to Domain Specific Languages (DSL), which is defined as a language engineered to work solely on a single program. This is highly relevant for blockchain, as Solidity was specifically designed for Ethereum, and therefore is a DSL. The authors then identify some common vulnerabilities in traditional smart contract code, and examine issues with traditional vulnerability checking techniques. Of these, one of the most important issues with traditional techniques is that the subset of bugs found are due to the strict predefined inputs that are used. The paper proposes that, through the use of Deep Learning, the input can be varied significantly to identify faults that the predefined static tests would otherwise not. The authors then propose a novel approach, which analysis the line level code and trains a Deep Learning Neural Network to understand the control paths and data transformations occurring in the code [12]. As an input to the model, to allow for the model to understand the code on a line level, the authors used an Abstract Syntax Tree (AST) structure, which relates variables to one another, marking their dependencies and transformations throughout the code. The author analyzed several Natural Language Processing techniques, and Recurrent Neural Networks, and eventually landed on using an LSTM network to train their model. They found that LSTM's outperformed most RNN models, and due to the vast variety in code syntax, the NLP techniques were unable to interpret many situations, since the code and inputs were inconsistently structured. Their results were quite accurate, but it is important to note that the results were tested against results from a traditional model that they were actually attempting to replace. If this paper could acquire a test set of vulnerabilities that were not acquired through the use of a traditional method, the results would be more poignant.

In 2021, Liu Z. et al. proposed a combining GNN and expert knowledge based machine learning model for detecting various smart contract vulnerabilities [25]. A graph neural network (GNN) is a deep learning method, where the principle is to perform inference on data described by graphs. In computer science, a graph is a data structure consisting of two components: nodes (vertices) and edges. Researches have proven that written programs can be converted to symbolic graph representation, without disrupting semantic relationship between programming elements. Thus, smart contract codes can be represented as contract graphs. In the experiment, ESC (Ethereum Smart Contracts) and VSC (VNT chain Smart Contracts) real world datasets (containing 320,000 contracts), where ESC was used to evaluate timestamp dependence vulnerabilities, while contracts from VSC is utilized for infinite loop vulnerabilities. The proposed model consists of two different parallel processes (Security pattern extraction and contract graph extraction) at the beginning, and the combining layer merged patterns in each section to find vulnerabilities, as shown in figure 3. First, a feed-forward neural network generates the pattern feature for extracting security patterns from the contract's source code. They have used an open-sourced tool to extract the expert patterns from smart contract functions. The second process (message propagation phase) is to create a GNN to achieve a contract graph. Inside the GNN model, nodes were the program elements (i.e., function), where edges represented the flow (i.e., next function to be executed) of each program elements. Later, unwanted nodes and edges are removed based on a node elimination strategy. As a preprocessing method, the authors casted rich control and data flow semantics of the source code into a contract graph. After this step, they designed a node elimination stage to highlight critical nodes by normalizing the graph. These two parallel processes were combined using vulnerability detection phase, where both extracted features are combined convolution and full-connected layer. In experiment, the proposed model is compared with non-ML-based security detection algorithms, namely Oyente, Myhrill, Smartcheck, Securify, and Slither. Each algorithm and the proposed model performed a search of several vulnerabilities (re-entrancy, timestamp dependence, and infinite loop vulnerabilities) of each function in the source code. The proposed algorithms (CGE) achieved 89% accuracy on finding re-entrancy and timestamp dependence type of vulnerabilities, and 83% accuracy on detecting infinite loop vulnerability [25].

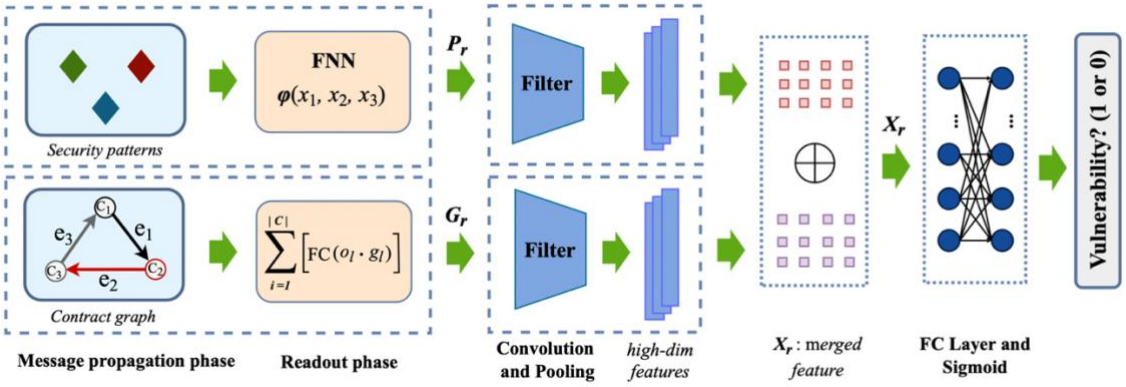

Figure 3. The process of vulnerability detection for the proposed combining GNN and expert knowledge model [25].

In 2021, Eth2Vec model is proposed to deficiency in current vulnerability detection tools when a code is rewritten. In programming languages, a code rewrite is reimplementing a source code's functionality without reusing it. When the smart contract codes are rewritten, detecting vulnerabilities become harder. The authors first converted each smart contract source code into EVM bytecodes. From the bytecode, the authors extracted only valuable information (i.e., function id, list of callee functions etc.) for vulnerability detection. As the last process, a neural network structure is used to catch any vulnerabilities in the source code. After testing the proposed model on 500 contracts, the Eth2Vec model was able to detect vulnerabilities with a 77% precision even though the contracts are rewritten.

In 2021, O. Lutz et al. [13] introduce yet another method of detecting vulnerabilities within smart contracts. The authors propose a solution entitled ESCORT, wherein they use a Deep Neural Network model to learn the semantics of the input smart contract, and learn specific vulnerability types based on the found semantics. The goal of the ESCORT model is to overcome the scalability and generalization limitations of traditional non-DNN models. Experimental results of this paper yielded an F1 accuracy score of 95% on six found vulnerability types, with a detection time of 0.02 seconds per contract. With such quick detection times, scalability is more easily achieved, satisfying one of the author's goals. Then, through the use of transfer learning, the ESCORT model slightly overcomes the issues found in other papers, such as Y. Xu or N. Lesimple's models [20-21], where newfound vulnerabilities can be realized by the model. Unfortunately, it is rather difficult to obtain interpretability from such models, and though new vulnerabilities may be found, understanding their cause remains to be exceedingly difficult.

In 2021, Sun et al. have attempted to detect the following vulnerabilities: re-entrancy, arithmetic issues (integer overflow/underflow) and timestamp dependence using machine learning [20]. As a common prerequisite step, some stack-operating instructions were truncated into more general forms (e.g., SWAP1, SWAP2, …, SWAPn. → SWAPx) to account for variations in instructions among different compilers. Following this, opcodes were separated into 9 categories based on their functions, as a label normalization step. As in [22] a word2vec transformation of the opcode sequences, preceding the convolutional layers, was performed. In addition to the pooling and softmax layers that commonly follow convolutional layers, this paper introduces an additional self-attention layer. The purpose of the self-attention layer is to create a connection between adjacent words in the obtained feature matrix since one-hot encoders that were used to encode each opcode instruction are just mere representatives and do not capture any functional similarity between them [20]. As a result, the word embedding process has been enhanced through the use of self-attention. When compared to the vulnerability detection performance of [22], they have both used a CNN but [22] used a word2vec embedding whereas this paper employed an attention mechanism, which is the likely reason that they obtained better results. obtained better results. The main improvement of the created model over the existing static analyzers such as Oyente and Mythril is that it can achieve comparable performance in much less time.

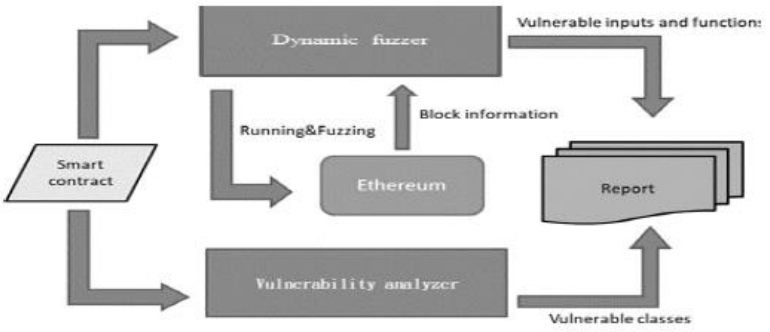

Figure 4. Model Overview of SoliAudit [22].

3.4 Classical Machine Learning Models

In this literature review, a classical machine learning model is referred to any method that does not employ deep or ensemble learning. They have a simpler internal architecture and require structured data to learn the pattern of the given input vector.

In 2019, Pouyan et al. employed popular supervised ML models to classify vulnerabilities in 1000 smart contracts [7]. The dataset was built collecting 1,013 smart contracts from Etherscan, where 80% was used for training and the remaining was used for testing purposes. In order to label each contract based on its source code's character, they used three different feature extraction techniques: abstract syntax tree (AST), control flow graph (CFG), and Static code analysis. The extracted features were grouped in two: features that represents execution path (e.g., function calls) and were directly added to the control flow graph. The authors used Slither and Mythrill to assign labels to each contract. 36 types of vulnerabilities were used to label each contract in test and train set. Vulnerability detection process was performed with common ML models: Support Vector Machine (SVM), Neural Network (NN), Random Forest (RF), and Decision Tree (DT). After training each model with training set, they were ranked based on the following evaluation metrics: accuracy, recall, F1, and precision. Due to the results, ML models were be able to identify 16 vulnerabilities among 36 with high performance. It was found that some specific ML models were more successful in finding certain vulnerabilities. For example, SVM model was successful of finding integer outflow, while NN achieved superior results detecting re-entrancy vulnerability. Due to the article's summary, it was proven that extracted features of smart contracts can be passed to any popular ML model for vulnerability detection. Also, it is important to note that static code analyzers' execution time (7,311 seconds) was drastically slower than any ML model (0.32 seconds) [7].

In 2021, a vulnerability and transaction behaviour-based detection is proposed [26]. In this work, the authors built a model that correlates malicious activities, and the vulnerabilities present in smart contracts. In respect to strength of the correlation unsupervised ML models (K-means and HDBSCAN) assign a severity score to each smart contract. The model was trained to detect suspects among benign smart contracts. The aim of the research was to test their hypothesis, which was "the transaction behavior is a more critical factor in identifying malicious smart contracts than vulnerabilities in the smart contract." Thus, they brought a different perspective to the literature of smart contracts vulnerability detection.

In 2021, Y. Xu et al.'s paper introduced two 'novel' smart contract vulnerability detecting approach using both a K-Nearest Neighbors (KNN) model and a Stochastic Gradient Descent (SGD) model [16]. Identifying some common vulnerabilities identified by traditional methods today, they attempt to use each of the machine learning models to identify eight of the most prominently recognized traditional vulnerability types: re-entrancy, arithmetic, access control, denial of service, unchecked low level calls, bad randomness, front running, and denial of service. As with N.Lesimple's paper [23], the input to their model uses an AST structure, allowing the model to gather line by line information about the smart contract code. The labels for the vulnerabilities were identified using traditional methods. The paper notes high accuracy, precision and recall, for four of the eight vulnerabilities. The other four did not have enough samples in the dataset, and the corresponding results were recognized as inconclusive. As with the N. Lesimple paper, the test set was created from results from using traditional methods, indicating that the authors were unable to illustrate how the KNN model differed from traditional techniques.

3.5 Ensemble Learning Models

Ensemble learning is a combination of multiple machine learning algorithms in an effort to increase the generalizability of the final outcome by fusing each model's individual outputs. Therefore, the objective of ensemble learning is to compensate other's weaknesses and ultimately achieve a greater performance.

In 2021, ContractWard model is proposed as a faster alternative for Oyente [27]. The dataset consisted of 49502 smart contracts, where each of them contained six possible vulnerabilities: integer overflow/underflow, transaction ordering dependency, call stack depth attack, timestamp dependency, and re-entrancy vulnerability. Each contract's source code is converted to opcodes. On average, a smart contract contains 4364 opcode elements with 100 types of opcodes in total. After the simplification process, there were only 50 opcode types left. Due to that reason, the authors wrapped opcodes with similar functionalities in a same category, and ultimately simplified features in the dataset. Later, they used n-gram technique (sliding window of binary-byte size) to track relations of each opcodes, since they assume that the operations have higher relation with its neighbors. Oyente was used to assign multi label to each contract. After the labeling process, the researchers encountered class-imbalance problem, due to rarity of some vulnerabilities. They employed synthetic minority oversampling technique to extend the number of minority class. The training process adopted 5 candidate ML models: eXtreme Gradient Boosting (XGBoost), Adaptive Boosting (AdaBoost), Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbour (KNN). In evolution stage, Micro-F1 and Micro-F1 (variations of F1 metric) is utilized to rank the predictors. The XGBoost model showed a robust performance by achieving over 96% F1, Micro-F1, and Macro-F1score.

In 2021, Esghie et al. proposed a novel monitoring framework (named as Dynamit) to uncover re-entrancy vulnerabilities in Ethereum smart contracts [28]. The novelty of the proposed model is that it does not observe the contract's code to make the prediction. Instead, their model relied on transaction metadata and balance data from blockchain system. Furthermore, additional to vulnerability detection, they were be able to get an execution trace that reproduces the attack. The framework has two main processes: monitoring transaction of the smart chain (the monitor) and detecting re-entrancy vulnerability (the detector). The monitor constantly listens to Ethereum blockchain client to fetch information about desired transactions by utilizing Web3js (Ethereum JavaScript API). Total of four different features are being extracted by the monitor: gas usage of transaction, contract 1 & 2 balance differences, and average call stack depth. The balance difference feature is essentially the balance of the contract's address before and after execution of the transaction. The average call stack depth feature represents the measure of recursive external calls invoking contract's function. In the study, the candidate predictors were popular ML models: Random Forest (RF), Naïve Bayes, Logistic Regression and K-Nearest Neighbours (KNN). At total, 105 transactions monitored by each model, where 53 of them were benign and 52 were harmful transactions and each transaction was associated with a label. After training and testing each models, RF model achieved the best classification score by detecting re-entracy vulnerability with 86% accuracy, 82% f1 score, and 74% recall. In order to test the validity of the Dynamit framework, the authors altered the smart contracts as well, where RF model (as a detector) achieved 94% accuracy, 93% f1 score and 94% recall.

In 2021, Y. Xue et al. [29] explores the concept of cross-contract vulnerabilities, which they posit are overlooked by most other vulnerability detection methods. Cross-contract vulnerabilities are "exploitable bugs that manifest in the presence of more than two interacting contracts" [29]. The complications of such an analysis arise when three or more contracts are interconnected, and this results in a highly non-trivial analysis to detect vulnerabilities amongst the connections. Rather than acquiring a fully labelled data set (of both benign and malware samples), the authors here focused on data paths that they knew were benign, and allowed their novel fuzzing framework approach, xFuzz, detect vulnerabilities in the data paths where malware might exist. The results presented show that their novel xFuzz approach detected 15 newly discovered vulnerabilities, that had not been detected by traditional static techniques. Furthermore, their approach was efficient, taking only 20% of the time than other fuzzing tools, while detecting almost twice as many vulnerabilities. The authors of this paper addressed several novel topics which prove the usefulness of certain ML techniques (tree-based models) when applied to detecting contract vulnerabilities: the efficiency of a reduced space fuzzing technique, and the effectiveness of cross-contract analysis, the latter of which is typically a highly complex issue, best handled by state-of-the-art ML techniques.

4. CONCLUSION

In today's world, blockchain has become an increasingly prevalent method of the distribution of information and currency. It's ability to transfer information both quickly and transparently, makes it an ideal method to track orders, transfer payments, and perform many other peer-to-peer transactions. Smart contracts are an integral component of all blockchain transactions. These contracts automatically execute the transaction of information, based on a set of immutable and publicly accessible instructions, so all parties can be confident in the outcome. Unfortunately, these contracts are subject to many kinds of vulnerabilities. These vulnerabilities have led to many malicious attacks, such as

TheDAO attack, which resulted in a loss of approximately $10 billion USD. As a result, methods to detect and correct these vulnerabilities are constantly being developed.

Some early methods, including Oyente and SmartCheck were introduced, and found a set list of vulnerabilities based upon predefined pattern detection methods. Though these early methods were successful in identifying the list of known vulnerabilities, the solutions were not robust to newer threats, as the ruleset upon which the analysis was based would have to be manually updated to reflect the newfound vulnerabilities. To adapt to the dynamic nature of vulnerabilities in smart contracts, machine learning techniques were explored, to attempt to outperform these traditional methods.

In this paper, we explored multiple machine learning techniques to identify vulnerabilities in smart contracts. Amongst them, deep learning algorithms were used in a variety of ways, with different types of input that allowed the models to identify vulnerabilities that the traditional methods sometimes could not. The input often consisted of a means to derive a mapping between the variables in the smart contract code, which allowed the models to detect the relationships between variables, and identify vulnerable structures and variable connections. The many reviewed uses of deep learning algorithms varied from one another by their design of the input structure, and these through their input design, the models were able to recognize different vulnerabilities in a myriad of ways. The reviewed methods utilizing classical machine learning methods and ensemble learning methods were largely developed in a similar fashion: to design a unique input structure for the model, to provide it with as much information relating to the structure and purpose of the smart contract as possible. The structure of the models themselves changed in each of the reviewed cases, though the models accuracy and effectiveness was largely dependent on the structure of the model's input. Examples of the input structures included raw opcode, sets of function inputs and outputs, Abstract Syntax Trees (AST), the transaction metadata and many others. One of the largest benefits of this wide variety in allowed input structures, is that implementer is not dependent on a specific type of information. For instance, if an individual cannot easily gain access to the smart contract source code, they can still utilize the transaction metadata to effectively detect vulnerabilities. Generally, it was found that the machine learning models outperformed the traditional methods in their efficiency, and matched their effectiveness.

Throughout this analysis, several faults were also noted for existing machine learning implementations. The first was the common labelling techniques used, which used traditional methods to label their input data. Most of the suggested models relied on supervised models, meaning labelled data was required for the model to train. To acquire the labels, to recognize the smart contracts as vulnerable or benign, traditional methods such as Oyente were used. As a result, the machine learning models were attempting to recognize vulnerabilities that were already interpretable through traditional methods. To improve this, manual detection of vulnerabilities should be implemented, or more unsupervised approaches should be explored. A second flaw found in many of the review approaches, was the inability for the models to provide interpretability. Due to the nature of machine learning models, sometimes the patterns found cannot be explained, and as a result, if a contract is found to be vulnerable, it becomes impossible to explain why. This is especially true when dealing with unsupervised approaches, as the models may identify vulnerabilities in code, but not have a defined labels to explain to any human why the contract was vulnerable. Since interpretability is an extremely important requirement when dealing with identifying vulnerabilities, further thought will have to be put into unsupervised approaches, to ensure the user knows why their contract is vulnerable. In conclusion, machine learning techniques provide many efficient and effective methods to identify smart contract vulnerabilities with a variety of input structures, but further research should be conducted in this space to provide interpretable solutions that outperform the traditional non-ML methods.

REFERENCES

[1] J. A. Kroll, I. C. Davey, and E. W. Felten, "The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries," 2013.

[2] M. Bartoletti and L. Pompianu, "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns BT - Financial Cryptography and Data Security," 2017, pp. 494–509.

[3] W. Zou *et al.*, "Smart Contract Development: Challenges and Opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, 2021, doi: 10.1109/TSE.2019.2942301.

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of Security and Trust*, 2017, pp. 164–186.

[5] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," *Bus. Inf. Syst. Eng.*, vol. 59, no. 3, pp. 183–187,

2017, doi: 10.1007/s12599-017-0467-3.
[6] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, “Decentralized Applications: The Blockchain-Empowered Software System,” *IEEE Access*, vol. 6, pp. 53019–53033, 2018, doi: 10.1109/ACCESS.2018.2870644.
[7] P. Momeni, Y. Wang, and R. Samavi, “Machine Learning Model for Smart Contracts Security Analysis,” *2019 17th Int. Conf. Privacy, Secur. Trust*, pp. 1–6, 2019, doi: 10.1109/PST47121.2019.8949045.
[8] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, “ReGuard: Finding Reentrancy Bugs in Smart Contracts,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 65–68.
[9] B. Jiang, Y. Liu, and W. K. Chan, “ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269, doi: 10.1145/3238147.3238177.
[10] P. Tantikul and S. Ngamsuriyaroj, “Exploring Vulnerabilities in Solidity Smart Contract.,” in *ICISSP*, 2020, pp. 317–324.
[11] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, “Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract,” *IEEE Access*, vol. 10, pp. 6605–6621, 2022, doi: 10.1109/ACCESS.2021.3140091.
[12] N. Lesimple, “Exploring Deep Learning Models for Vulnerabilities Detection in Smart Contracts,” 2020.
[13] O. Lutz *et al.*, “ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning.” 2021.
[14] X. Liang Yu, “Oyente,” *GitHub*, 2017.
[15] E. Marchenko, “SmartCheck,” *GitHub*, 2020, [Online]. Available: https://github.com/smartdec/smartcheck.
[16] Y. Xu, G. Hu, L. You, and C. Cao, “A Novel Machine Learning-Based Analysis Model for Smart Contract Vulnerability",” *Secur. Commun. Networks*, vol. ID 5798033, p. 12, doi: 10.1155/2021/5798033.
[17] B. Wang, H. Chu, P. Zhang, and H. Dong, “Smart Contract Vulnerability Detection Using Code Representation Fusion,” in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, 2021, pp. 564–565, doi: 10.1109/APSEC53868.2021.00069.
[18] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, “EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 127–137, doi: 10.1109/ICPC52881.2021.00021.
[19] S. Bistarelli, G. Mazzante, M. Micheletti, L. Mostarda, D. Sestili, and F. Tiezzi, “Ethereum smart contracts: Analysis and statistics of their source code and opcodes,” *Internet of Things*, vol. 11, p. 100198, 2020, doi: https://doi.org/10.1016/j.iot.2020.100198.
[20] Y. Sun and L. Gu, “Attention-based Machine Learning Model for Smart Contract Vulnerability Detection,” *J. Phys. Conf. Ser.*, vol. 1820, p. 12004, 2021, doi: 10.1088/1742-6596/1820/1/012004.
[21] W. J.-W. Tann, X. Han, S. Gupta, and Y.-S. Ong, *Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Vulnerabilities*. 2018.
[22] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, “SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing,” in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 458–465, doi: 10.1109/IOTSMS48152.2019.8939256.
[23] A. K. Gogineni, S. Swayamjyoti, D. Sahoo, K. K. Sahu, and R. Kishore, “Multi-Class classification of vulnerabilities in Smart Contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing,” *CoRR*, vol. abs/2004.0, 2020, [Online]. Available: https://arxiv.org/abs/2004.00362.
[24] C. Xing, Z. Chen, L. Chen, X. Guo, Z. Zheng, and J. Li, “A new scheme of vulnerability analysis in smart contract with machine learning,” *Wirel. Networks*, Jul. 2020, doi: 10.1007/s11276-020-02379-z.
[25] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, *Smart Contract Vulnerability Detection using Graph Neural Network*. 2020.
[26] R. Agarwal, T. Thapliyal, and S. K. Shukla, “Vulnerability and Transaction behavior based detection of Malicious Smart Contracts,” *CoRR*, vol. abs/2106.1, 2021, [Online]. Available: https://arxiv.org/abs/2106.13422.
[27] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, “ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts,” *IEEE Trans. Netw. Sci. Eng.*, vol. 8, pp. 1133–1144, 2021.
[28] M. Eshghie, C. Artho, and D. Gurov, “Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning,” in *Evaluation and Assessment in Software Engineering*, 2021, pp. 305–312, doi: 10.1145/3463274.3463348.

[29] Y. Xue *et al.*, “Machine Learning Guided Cross-Contract Fuzzing,” *CoRR*, vol. abs/2111.1, 2021, [Online]. Available: https://arxiv.org/abs/2111.12423.