IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 8 January 2024, accepted 30 January 2024, date of publication 8 February 2024, date of current version 16 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3364351

APPLIED RESEARCH

An Integrated Smart Contract Vulnerability Detection Tool Using Multi-Layer Perceptron on Real-Time Solidity Smart Contracts

LEE SONG HAW COLIN[1], (Member, IEEE), PURNIMA MURALI MOHAN[1], (Member, IEEE), JONATHAN PAN[2], (Member, IEEE), AND PETER LOH KOK KEONG[1], (Senior Member, IEEE)

[1]Infocomm Technology Cluster, Singapore Institute of Technology, Singapore 138683
[2]Disruptive Technologies Office, Home Team Science and Technology Agency, Singapore 138507

Corresponding author: Purnima Murali Mohan (purnima.mohan@singaporetech.edu.sg)

This work was supported by the Singapore Ministry of Education (MoE) Grant, Singapore Institute of Technology (SIT).

ABSTRACT Smart contract vulnerabilities have led to substantial disruptions, ranging from the DAO attack to the recent Poolz Finance. While initially, the smart contract vulnerability definition lacked standardization, even with the advancements in Solidity, the potential for deploying malicious contracts to exploit legitimate ones persists. The Abstract syntax tree (AST), opcodes, and control flow graph (CFG) are the intermediate representations for Solidity contracts. In this paper, we propose an integrated and efficient smart contract vulnerability detection algorithm based on Multi-layer perceptron (MLP). We use feature vectors from the Opcodes and CFG for the machine learning (ML) model training. The existing ML-based approaches for analyzing the smart contract code are constrained by the vulnerability detection space, significantly varying Solidity versions, and no unified approach to verify against the ground truth. The primary contributions in this paper are 1) a standardized pre-processing method for smart contract training data, 2) introducing bugs to create a balanced dataset of flawed files across Solidity versions using AST, and 3) standardizing vulnerability identification using the Smart Contract Weakness Classification (SWC) registry. The ML models employed for benchmarking the proposed MLP, and a multi-input model combining MLP and Long short-term memory (LSTM) in our study are Random forest (RF), XGBoost (XGB), Support vector machine (SVM). The performance evaluation on *real-time* smart contracts deployed on the Ethereum Blockchain show an accuracy of up to 91% using MLP with the lowest average False Positive Rate (FPR) among all tools and models, measuring at 0.0125.

INDEX TERMS Blockchain, ethereum, machine learning, multi-layer perceptron, real-time smart contracts, solidity smart contracts, vulnerability analysis and detection, code analysis, software testing.

I. INTRODUCTION

In 2022 the amount of funds controlled by smart contracts was worth around USD 1750 million and it is set to reach USD 9850 million by 2030 [1]. The gaining popularity of smart contracts is due to its autonomy, trust, and secure environment and this has given birth to what we know as Decentralised applications (*Dapps*) today. Dapps are applications that have their logic written in smart contracts and deployed to the blockchain. Dapps covers a wide range of use cases from logistics to finance which enables day to day and monitoring of usage. The transparency and traceability characteristics of a blockchain attract not only private companies but also keen government agencies. For example, a permissioned or private blockchain can be used to manage the flow of currency [2]. Blockchain also finds application

The associate editor coordinating the review of this manuscript and approving it for publication was Huiyan Zhang.

in sharing data between heterogeneous devices using smart contracts. A framework by [3] suggests using smart contracts for recording data in the cloud, for data security, and accountability of Internet of Things (IoT) devices. Another use case for IoT smart contracts is unmanned aerial vehicles (UAVs) [4] that transform a centralized trusted authority into a secure decentralized network. However, these works use smart contracts as authentication mechanisms for IoT and assume the inherent security of smart contracts.

Smart contracts have not been immune to hacks, since early 2016 when major hacks started to surface and gain media attention. These hacks are not only harmful to the industry but also cast a large shadow over the longevity of blockchain applications. In 2022, approximately 1.9 billion USD have been lost to various hacks by exploiting vulnerabilities in smart contract logic and manipulation of human errors in smart contracts [5]. To add on, specifically, Poolz Finance suffered a major *arithmetic overflow* hack, where one of its methods for pool creation contains a manual summing of token count which resulted in losing USD 6, 650, 000 [6], [7].

There have been attempts to prevent such hacks by creating contract standards and defining vulnerabilities. OpenZeppelin [8], Consensys [9] and a trail of bits [10] are some of the front runners on the quest to support developers and overcome potential security vulnerabilities.

In terms of standardization, smart contract vulnerabilities have been loosely defined since its emergence. There have been different flavors namely — the Smart Contract Weakness Classification (SWC) registry [11], DASP [12] and Crytic's static analyzer, *i.e.*, the Slither's detector documentation [13]. These definitions cover most of the notable vulnerabilities, however, there is an existing gap in terms of compliance with any security standards and coverage of the entire smart contract vulnerability space constrained by the varying solidity versions and ever-emerging smart contract logic bugs. These vulnerabilities, often resulting from human errors and version changes, underscore the need for more robust and reliable security measures in smart contracts. While such vulnerabilities can be detected by software verification and validation tools using static [13], [14], dynamic [15], and formal verification [14], [16], each method has its own limitations. In the work done by [17], the authors have conducted an extensive test on software verification and validation tools and concluded that there is no single analysis tool that can detect all smart contract vulnerabilities. Not only that new vulnerabilities cannot be detected if it was not predefined within the tool, but also existing vulnerability detection had significant *false positive rates*. These findings therefore advocate the use of a Machine Learning (ML) approach that can be dynamically trained to newer smart contract bugs and solidity versions while reducing the false positive rate.

For any ML model the key factors that decide its reliability of that model are, (i) the origin of the dataset used for training that model, (ii) the quality and correctness of the training dataset, (iii) scalability and run-time of the model, and (iv) standardized verification and validation of the model. Recent works that have claimed to have sourced data from reliable sources [18], [19], [20], are still based on the validation of software verification and validation tools such as Mythril [14], Slither [13], and Oyente [16] to determine the ground truth information about the smart contracts used for training. Firstly, the main concern with such reliance on third-party software verification tools for the training dataset is that it is prone to inaccuracies that are inherent to these software verification tools. Secondly, to introduce vulnerabilities into the training dataset, a synthetic data generation method using the Synthetic Minority Oversampling Technique (SMOTE) is being used [19], [20]. However, synthetic data can never be a true representation of a truly vulnerable dataset. Not only does it not keep updated with the significantly varying Solidity versions, but also leads to an imbalance in the training dataset due to the way it is implemented. Thirdly, some existing works simply skip the pre-processing steps (that ensures the quality and correctness of the training dataset) [18] which makes the solution practically not useful — especially when the training data set that comes across different solidity versions, contains commented codes with code-like syntax. This becomes a prime concern since the *solc* compiler cannot differentiate code-like commented syntax — which is crucial for bug-insertion algorithm. Also, the existing bug-injection tools [21] suffer from practical issues such as solidity version, syntax based on solidity version, no exception, and no control over bug injection logic (i.e., the existing tools dump all bugs in a single smart contract). Motivated by these practical concerns, our proposed solution injects known vulnerability patterns into clean contracts by ensuring a clear indication of a vulnerability bug type in each contract while developing a standardized pre-processing method to generate a balanced and good quality training dataset and validate against well-defined vulnerability standards.

In this paper, we will be introducing an ML approach that uses a *runtime opcode extraction algorithm* for feature extraction and a *trigram-based* method for vectorization. We reference the SWC [11] registry for smart contract vulnerability categorization as it has been one of the most well-defined mappings while loosely coupled with the Common Weakness Enumeration (CWE) [22]. To test the efficacy of our solutions and perform benchmarking, we will employ Mythril, Slither, and an integrated tool known as MythSlith. We have developed MythSlith — which is a tool that integrates Mythril and Slither to increase the coverage of smart contract vulnerability detection space. These tools will be used to compare against the ML model trained with the bug-injected dataset. Our contributions in this paper are summarized as follows:

- We developed a standardized pre-processing algorithm for cleaning smart contract training data to address

the limitation of the *solc* compiler not being able to differentiate code from commented code-like syntax.
- We developed a practical bug injection algorithm to create a balanced dataset across Solidity versions using the Abstract syntax tree on verified smart contracts that were cleaned using the proposed pre-processing algorithm.
- We model an MLP framework and a multi-input model based on MLP and LSTM for smart contract vulnerability detection. The proposed framework scales up the smart contract vulnerability space by utilizing opcodes and Control Flow Graph (CFG) extracted during model training. Before vectorization, a simplification method is applied to the opcodes to decrease dimensionality (reduce the running time) and eliminate contract-specific hexadecimal values.
- We thoroughly analyze the time complexity of the proposed algorithms and the running time for bug detection.
- For MLP framework performance benchmarking, we integrate the well-known software verification tools, Mythril and Slither, to develop an experimental tool known as MythSlith which has an increased smart contract vulnerability detection space than the individual tools. The machine learning models show a superior performance in vulnerability detection than existing Software verification tools with the lowest false positive rate of up to 0.0015.
- We verify the results against the standardized vulnerability identification — Smart Contract Weakness Classification (SWC) registry as a common analysis platform.

The rest of the paper is organized as follows: Section II provides a background of smart contract intermediate representation, smart contract vulnerabilities, and types of software validation techniques used. Section III presents a literature review of the recent works on smart contract vulnerability detection space. In Section IV, we illustrate the proposed methodology and framework for feature extraction to model training algorithms. Section V reports the experimental findings and inferences made on the efficiency of the proposed methods. Section VI will discuss the findings, some possible alternatives, and future development before we conclude our work in Section VII.

II. BACKGROUND

In this section, we provide an overview of the vulnerabilities and tools that will be used in the proposed algorithm for smart contract vulnerability detection.

A. SMART CONTRACTS

Smart contracts are programs written to be executed on the blockchain. Designed to be autonomous, self-sufficient and expected to do their written or agreed-upon task in code without interference. They are considered to be the key to a decentralized system. However, being in the eye of the public blockchain any vulnerabilities could lead to a huge amount of financial loss.

```
mapping(address => uint) userBalance_re_ent5;
function withdrawBalance_re_ent5() public {
    if(!(msg.sender.send(userBalance_re_ent5[msg.sender])))
    {
        revert();
    }
    userBalance_re_ent5[msg.sender] = 0;
}
```

FIGURE 1. Reentrancy snippet.

```
mapping(address => uint) balances_intou2;

function transfer_undrflow2(address _to, uint _value)
    public returns (bool)
{
    require(balances_intou2[msg.sender] - _value >= 0);
    balances_intou2[msg.sender] -= _value;
    balances_intou2[_to] += _value;
    return true;
}
```

FIGURE 2. Arithmetic snippet.

Smart contracts can also be represented in different forms with the help of the Solidity compiler. Some forms are opcodes, abstract syntax tree (AST), and Control Flow Graph (CFG).

Opcodes also known as operation code, are instructions for the Ethereum virtual machine (EVM) to execute any sequential and conditional actions. The complete list of opcodes with descriptions can be found in Ethereum's yellow paper [23].

Abstract Syntax Tree is a hierarchical tree representation of the synthetic structure of the source code. Each section is represented as nodes, which construct the details of the real syntax. Such representation is commonly used for syntax checking, semantic analysis, code generation, and code optimization.

Control Flow Graph is the representation of program flow from the context of the stack. This is derived from the opcodes, where a bunch of opcodes is broken down into basic blocks by flow conditions such as `JUMP`, `JUMPI`, `REVERT`, etc.

B. TYPES OF BUGS

Smart contract vulnerabilities are often caused by oversights during the programming stage, these bugs may seem harmless when written but can potentially cause huge financial loss. The 7 vulnerabilities below are chosen as they are commonly found in other studies and they have become the foundation for vulnerabilities in both empirical and ML-based approaches [19], [21], [24], [25]. The availability of predefined bug snippets in [21] also helps cut down the amount of development time.

Reentrancy was first uncovered in 2016 when a large sum of money was stolen in the DAO contract (Fig 1). This was primarily caused by the action of sending cryptocurrency to an external account and updating it only after it was sent.

```
function bug_unauth_send5() payable public
{
      msg.sender.transfer(1 ether);
}
```

FIGURE 3. Unauthorized send snippet.

```
function transferTo_txorigin3(address to, uint amount,
    address owner_txorigin3) public
{
  require(tx.origin == owner_txorigin3);
  to.call.value(amount);
}
```

FIGURE 4. Tx origin snippet.

```
function bug_tmstmp5() view public returns (bool) {
  return block.timestamp >= 1546300800;
}
```

FIGURE 5. Timestamp dependency snippet.

```
bool claimed_TOD10 = false;
address owner_TOD10;
uint256 reward_TOD10;
function setReward_TOD10() public payable {
    require (!claimed_TOD10);

    require(msg.sender == owner_TOD10);
    owner_TOD10.transfer(reward_TOD10);
    reward_TOD10 = msg.value;
}

function claimReward_TOD10(uint256 submission) public {
    require (!claimed_TOD10);
    require(submission < 10);

    msg.sender.transfer(reward_TOD10);
    claimed_TOD10 = true;
}
```

FIGURE 6. Transaction order dependency.

This sequence of events might seem normal for traditional software code, however, in the case of Solidity, a fallback function of the external account can re-trigger the same sending function again before an update can happen within the original contract. This could result in an indirect recursive function call.

Arithmetic vulnerabilities, more commonly known as Overflow-Underflow (Fig 2). Overflow occurs when an operation tries to add to a variable that is already at its maximum possible value. Without any sort of guard, this variable will overflow and back to 0 or the minimum possible value. As opposed to Overflow, Underflow happens when an operation tries to subtract a variable when it is at the minimum possible value, resulting in the value jumping to the maximum possible value. This vulnerability can lead to severe security issues as hackers can use this behavior to alter account balance or change ownership of contract.

Unauthorized send arises when there is no access control for a function that requires an access check (Fig 3). Such a function may contain withdrawal or reward disbursement functionality.

Transaction origin also known as `tx.origin`, arises from the misuse of the global variable 'tx.origin' of Solidity (Fig 4). In all transactions there is an origin and a sender. Origin is the address that started the chain of calls while

```
bool public payedOut_unchk9 = false;

function withdrawLeftOver_unchk9() public {
    require(payedOut_unchk9);
    msg.sender.send(address(this).balance);
}
```

FIGURE 7. Unhandled exceptions.

the sender is the address that initiated the current call. Tx origin vulnerability occurs when a contract uses 'tx.origin' to authenticate a user rather than 'msg.sender', which refers to the current immediate caller. This is particularly dangerous when a `tranferOwnership` function uses 'tx.origin' for authentication.

Timestamp dependency arises when a contract uses block variables such as block hash, timestamp, number, difficulty, gaslimit and coinbase to perform critical operations (Fig 5). These operations are generation of random numbers and time critical applications such as auction. This is in particular dangerous because miners gets to choose the block's timestamp.

Transaction Order Dependency is a type of vulnerability where the sequence of calling transactions can impact the final outcome of the application (Fig 6). This arises when a state is altered based on the order of incoming transactions, which can lead to unintended exploitation.

Unhandled Exceptions occurs when checks on send, transfer, or call are not done (Fig 7). This is important because calls can fail and intended changes will be reverted. Some of the reasons for failure can be, out-of-gas exceptions and wrong arithmetic operations such as zero division errors.

III. LITERATURE REVIEW

In this section, we will discuss currently available software verification and validation tools for smart contract detection, as well as works that use a machine learning approach. Our review will cover the data sampling techniques, feature extraction methods, and smart contract vulnerability standards. An overview of the comparison can be found in Table 1.

A. SOFTWARE VERIFICATION AND VALIDATION TOOLS

In recent years, a number of analysis tools have been introduced and they can be categorized into 3 different types, Static, Dynamic, and Formal verification. Static tools rely on the static information of the code to derive a prediction without executing the program [27]. Some of those features extracted are the abstract-syntax tree (AST), compiled bytecode, and opcodes. Dynamic tools analyze a running program, one such example is the Fuzzer [28]. Formal Verification tools rely on the mathematical definition and use a solver such as Z3 to resolve the derived formula [29].

Among the software verification and validation tools for smart contracts are Slither [13], Mythril [14], and DefectChecker [30] for static verification; Manticore [15] for dynamic analysis; and Oyente [16], Mythril, and DefectChecker for formal verification. Although these tools

TABLE 1. Comparison with existing works.

Reference	Data Source Labeling	Pre-processing		Mapping to SWC	Bug Injection Validation
		Solidity Version	Comments removal		
[18]	Mythril, Slither	✓	✗	✗	✗
[19]	Oyente	✗	✗	✗	✗
[20]	Mythril, Slither	✗*	✗	✗	✗
[26]	Oyente, Slither, DefectChecker	✗	✓	✗	✗
[21]	Bug Injection	✗	✗	✗	✗
Our work	Bug Injection	✓	✓	✓	✓

* Author stated as pre-processing done but with no steps/details

have been instrumental in numerous audits, they are constrained by the predefined patterns of each bug. Should new bugs emerge, an expert update would be necessary.

B. MACHINE LEARNING BASED VULNERABILITY DETECTION

As the name suggests, ML method is used to construct a set of decisions from the features of the data to make a logical conclusion of a trend or classification. Such a set of decisions is known as a model and it can be done with supervised or unsupervised learning. Supervised learning is an algorithm that learns with labeled data while unsupervised will suggest clusters of possible outcomes and then deriving of the outcome will depend largely on the feature set.

1) DATA SAMPLING

As discussed in Section I, data integrity is an integral part of ML model training but existing works from [18], [19], and [20] did not have a cleaning or pre-processing step to ensure contracts are clean. While [26] has done checks using three software validation tools namely, Slither, Oyente, and DefectChecker to ensure contracts are properly labelled and also removed if any error were present in the tool's output, in addition, smart contracts without version numbers were also removed. However, the labeling of data relies on software validation tools, which face the same issue as previous works, i.e., the training dataset is prone to inaccuracies inherent to these tools.

Whereas in our approach, bug injection ensures identified bugs to be injected. However, bug injection from Solidifi [21] lacks post-injection error checking, leading to the creation of an erroneous dataset. Moreover, the code from [21] did not anticipate a situation where the bug count is less than the available injection location. To accommodate such scenarios, a recursive function was incorporated to check on the bug count and the number of available locations. In addition, we have also incorporated a validation and error-handling mechanism into the proposed bug injection algorithm.

2) FEATURE EXTRACTION

Using opcodes for feature extraction is one of the more popular methods as it is agnostic to the expert pattern and presents a clear path to identify any malicious act. Abstract Syntax Tree (AST) is also one of the common methods as it contains useful semantic information. Reference [19] uses a simplified opcode followed by Bigram for vectorization, while [18] uses features from AST, [20] and [26] uses a mixture of AST and simplified Opcode with Bigram.

While traditional models typically process data in a single common format, this does not preclude the combination of vectorized data from various features. Works of [20] and [26] have both mixed their feature embeddings and can produce a good amount of variance for classification. However, in our proposed work aside from the classical models, we will implement a multi-model approach that allows features of different shapes to collaborate effectively.

C. VULNERABILITY STANDARDS

Vulnerability standards in the smart contract field are not yet prevalent, resulting in a spew of different definitions by different organizations [11], [12]. This is not healthy for the industry and will hinder further development. It is also clear from existing works [18], [19], [20], [26], that no vulnerability standards were taken up. Consequently, in our proposed work, we have selected the SWC Registry [11] as the vulnerability standard to provide a clear definition.

In the proposed solution, opcodes will be used alongside CFG features. Trigram is used instead as it captures more context than Bigram or unigram. Rather than using AST, CFG is chosen because it contains flow information which AST does not. Simplification of opcodes is done but different from previous works, rather than replacing the entire set of PUSH, DUP, and SWAP opcodes with a constant, we will leave the first five numbers untouched, allowing more variance to be captured. In addition, we have also removed all hexadecimal values to prevent any contract-specific values which could be learned by the models.

This review has highlighted the need for expert knowledge to define new bugs for current software verification and validation tools, significant variations in data sampling and feature extraction methods, and the absence of vulnerability standards. While each work does give clear definitions of their selected vulnerabilities, there is no compliance. Data sampling from [26] have implemented a robust method using validation tools for labeling, others have not, raising concerns about data integrity. Opcode is a popular representation for feature extraction and mixing of features is a popular method as seen in works of [20] and [26] but a multi-model approach

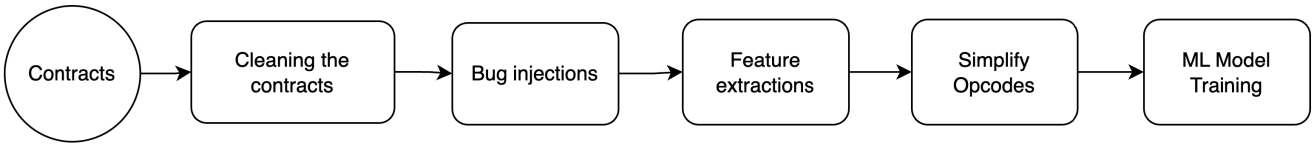


FIGURE 8. Pre-processing Flow Diagram for the ML model training.

has yet to be attempted. The disarray in vulnerability standards indicates the need for a single, universally adopted standard to streamline the development process. The insights gained from this review provide a good foundation for developing a robust and reliable smart contract vulnerability detection method.

The research gap is summarized in Table 1 highlighting the novelty of the paper. In this paper, we perform standardized pre-processing steps, by introducing both erroneous solidity version exclusion and code-like comments removal which could generate an incorrect AST. For data source labeling we use bug injection in our work as it provides a reliable ground truth. This was not done in the referenced existing works in Table 1. While the work in [21] uses bug injection, it does not validate the bug injection nor include any pre-processing steps. The most recent work that performs pre-processing by removing code-like comments is found in [26]. However, it does not verify incorrect solidity versions (during pre-processing) while relying on third-party tools (such as Mythril, Slither, Oyente, DefectChecker, etc.) to label their datasets.

IV. METHODOLOGY

The approach to this research will be detailed in this section in the following sequence: IV-A Preparation of dataset, IV-B Feature extraction, IV-D Machine Learning Models for Classification, IV-E Multi-Model Approach, IV-F Design of MythSlith, IV-G Challenges. A flow diagram of the end-to-end pre-processing steps that lead to the ML model training is illustrated in Figure 8.

A. PREPARE DATASET FOR ML TRAINING

1) DATASET

For any ML model to be efficient and usable, generating an error-free and practical dataset is important to achieve good accuracy and false positive rate. As a first step, the clean dataset was initially sourced from the smart contract sanctuary [31] — an open-sourced repository and we validated the ground truth by compiling each Solidity file using the Solidity compiler (solc) to ensure no errors were present before any bug injection was performed. The version of the compiler used for each file is determined with a JSON file provided by [31], which consists of the specific version used when the smart contract was actually deployed to the mainnet.

The second step is to use the validated clean dataset and inject with bugs defined by Solidifi [21] — which is an automated bug injection tool that checks for potential locations using AST tree to inject a set of predefined bugs. However, Solidifi cannot be directly used to inject bugs since it suffers drawbacks such as there is (i) no solidity version check, (ii) no syntax check, (ii) no exception for bugs, and more importantly, (iv) it injects all the predefined bugs to a single solidity smart contract which is not a practical scenario to train the ML model. We hence propose two pre-processing algorithms and a bug injection algorithm below to mitigate the drawbacks of Solidifi.

TABLE 2. Variables used in the bug injection framework.

Variable	Description
f_c	The file contents. This is the data that will be manipulated by the algorithm.
f_c^*	new file contents after bug injection
b_{selected}	selected bug
nb	The number of bugs. This is one of the factors that determines how many times the file will be modified.
$\mathcal{L}_{\text{fDef}}$	List of functions obtained from AST.
f_{selected}	Selected function from list of functions obtained from AST
B	Set that represents all the bugs
B_{used}	Set that represents all used bugs
B_{unused}	Set that represents all unused bugs
AST	Abstract Syntax Tree of the current file content
$SOLC$	Solidity compiler

2) BUG INJECTION

The goal of employing the bug injection technique is to imitate the introduction of bugs by developers [21] in the smart contract logic. The number of bug snippets is determined by a predefined *bug density*. The bug density is defined as the number of vulnerable lines of code per clean smart contract. Refer to Table 2 for the variables used in the Bug injection algorithm. For example, for every 100 line of code, when we insert 1 line of bug, then the bug density for that smart contract will be 1%. By setting the bug density, we are able to have an even spread of bugs within the entire dataset used for training the ML model. To be uniform across the smart contract vulnerability space, we represent each bug snippet as a function.

The process of injecting bugs has the following steps:

Step1: Pre-process the source file using Algorithm (1)
Step2: Obtain source attribute information by generating the abstract syntax tree (AST)

Algorithm 1 Pre-Processing Procedure

```
procedure PreProcess(file_path, output_directory)
    lines ← ReadLines(file_path)
    new_lines ← []
    inside_comment_block ← False
    is_tgt_with_start_end_block ← False
    is_inside_unique_comment_block ← False
    for line ∈ lines do
        if "/*/" in line then
            Toggle is_inside_unique_comment_block
            Continue
        end if
        if is_inside_unique_comment_block then
            Continue
        end if
        if "/*" and "*/" in line and "/*/" not in line then
            Remove inline comment in line
        end if
        if "/*" in line and "//***" not in line then
            inside_comment_block ← True
            Remove text after "/*" in line
        end if
        if "*/" in line then
            inside_comment_block ← False
            Remove text before "*/" in line
        end if
        if inside_comment_block then
            if is_tgt_with_start_end_block then
                is_tgt_with_start_end_block ← False
                Append line to new_lines
            end if
            Continue
        end if
        if "http://" or "https://" in line then
            Append line to new_lines
            Continue
        end if
        if "//" in line then
            Remove text after "//" in line
        end if
        Append line to new_lines
    end for
    file ← Join(new_lines)
    Write file to output_directory
    return file
end procedure
```

Step3: Filter the function definition within the smart contract definition that is generated by the AST using Algorithm 2

Step4: Generate the number of bugs to be injected per Solidity file based on a pre-defined bug density

Step5: Randomly select a function location (L_{fDef}) from the AST node and randomly choose a bug ($b_{selected}$) from the list of smart contract bugs to insert the bug to the identified random location using the Algorithm 3

Algorithm 2 Filtering Function Definition From AST

```
procedure F(AST)
    Initialize L_fDef ← ∅
    for n ∈ N do
        t ← τ(n)
        if t = "FunctionDefinition" then
            L_fDef ← L_fDef ∪ {n}
        end if
    end for
    return L_fDef
end procedure
```

a: PRE-PROCESSING SOLIDITY FILES

Cleaning Solidity files is a crucial pre-processing step, as the solc compiler cannot distinguish between commented code and code-like syntax. This distinction is especially vital in our implementation, as it impacts the injection process and, consequently, influences the false positive rates in the bug classification process. The details of the pre-processing algorithm are outlined in Algorithm (1).

Similar to the approach used in [21] bug locations are derived from AST nodes of the given Solidity files. The injection process is shown in detail in Algorithm (2) wherein the fuction is filtered from the AST tree and in Algorithm (3) where it takes in the file contents, number of bugs, the list of all bugs to uniquely insert as inputs to recursively insert the bugs in randomly selected function location until one of the below conditions are met.

- the function locations are exhausted
- the bug density condition is satisfied
- all available bug snippets are exhausted

The above condition checks will only be done after a compilation check using solc after every bug injection. This ensures that the new file is free of errors, i.e., an error-free dataset for training the ML model.

b: AST INSTANCE

When defining the AST instance in this section, we will only take the attribute of interest into consideration. Let T be the set of node types, including 'FunctionDefinition', 'PragmaDirective', 'ContractDefinition', 'VariableDeclaration', etc. Let $\mathcal{N}$ be the set of nodes, where node $n \in \mathcal{N}$ and n has a type from T. Let S be the set of all possible 'src' attributes, representing the location of a particular node in the code. We define the below functions to define the AST tree instance per solidity file.

- $\tau : \mathcal{N} \rightarrow T$: Maps each node to its type (corresponding to the 'nodeType' attribute).
- $C : \mathcal{N} \rightarrow 2^{\mathcal{N}}$: Maps each node to its child nodes, capturing the hierarchical structure of the code.

Algorithm 3 Bug Injection Algorithm

procedure InjectBugRecursively(fc, nb, B, B_{used})
  $AST \leftarrow SOLC(f_c)$
  $\mathcal{L}_{\text{fDef}} \leftarrow F(AST)$
  $B_{\text{unused}} = B - B_{\text{used}}$
  $f_{\text{selected}} \leftarrow R(\mathcal{L}_{\text{fDef}})$
  if ($f_{\text{selected}} = \text{None}$) **or**
  ($|B_{\text{used}}| = nb$ **and** $|B_{\text{unused}}| = 0$) **then**
    Return fc
  end if
  $b_{\text{selected}} = R(B_{\text{unused}})$
  $f_c^* = SliceIn(f_c, b_{\text{selected}}, f_{\text{selected}})$
  $B^*_{\text{used}} = B_{\text{used}} \cup \{b_{\text{selected}}\}$
  Comment: Recursive call to continue bug injection
  $InjectBugRecursively(f_c^*, nb, B, B^*_{\text{used}})$
end procedure

- $\sigma : \mathcal{N} \rightarrow S$: Maps each node to its 'src' attribute.

The AST is then represented as a 4-tuple containing the nodes, their types, child nodes, and source locations.

$$\text{AST} = (\mathcal{N}, \tau, C, \sigma)$$

c: FUNCTION EXTRACTION

Next, we define the function that extracts only the'Function Definition' from the AST nodes. Define a function $F : \text{AST} \rightarrow \mathcal{L}$ to extract nodes with the 'nodeType' of 'FunctionDefinition' from the AST. The function extraction step operates as follows:

- It takes the AST as input, where the AST is represented as $\text{AST} = (\mathcal{N}, \tau, C, \sigma)$.
- It iterates through the nodes in $\mathcal{N}$, checking the 'nodeType' attribute for each node using the function $\tau : \mathcal{N} \rightarrow T$.
- If the 'nodeType' of a node is 'FunctionDefinition', it adds the node to the resulting list.
- It returns the list of nodes with the 'nodeType' of 'FunctionDefinition' as the output.

$$\mathcal{L}_{\text{fDef}} \leftarrow F(AST)$$

d: BUG DENSITY

Selection of the number of bugs is done with a bug density value, which is a predefined ratio of 0.01, and each clean contract will have at least 1 bug inserted:

Let $\mathcal{S}$ represent the number of lines of code in the file, and let δ be the predetermined constant ratio representing bug density. The variable nb, which denotes the number of bugs to be injected, is given by:

$$nb = \max(1, \lfloor \mathcal{S} \times \delta \rfloor)$$

Here, $\max(1, \ldots)$ ensures that the minimum number of bugs is at least 1, and $\lfloor \ldots \rfloor$ represents the floor function, rounding down to the nearest integer.

e: INJECTION OF SMART CONTRACT BUGS

With each iteration of bug injection, used bugs will be tracked, hence no bugs will be repeated which will result in compilation failure. To represent this, we will let B be the set of all available bug files, and let B_{used} be the set of bug files that have already been used. The set of remaining bug files B_{unused} can be represented as:

$$B_{\text{unused}} = B - B_{\text{used}}$$

Here, $B - B_{\text{used}}$ denotes the set difference, which results in a new set B_{unused} containing all the elements that are in B but not in B_{used}.

A random function definition node, which has the source location, will be selected for each bug. This is represented by f_{selected} and it is selected from $\mathcal{L}_{\text{fDef}}$:

$$f_{\text{selected}} = \begin{cases} R(\mathcal{L}_{\text{fDef}}) & \text{if } \mathcal{L}_{\text{fDef}} \neq \emptyset \\ \text{None} & \text{otherwise} \end{cases}$$

Here, $R(\mathcal{L}_{\text{fDef}})$ represents the random selection of a function definition node from $\mathcal{L}_{\text{fDef}}$. If $\mathcal{L}_{\text{fDef}}$ is empty, f_{selected} will be set to None.

Similar to how a function definition node is selected, bug is also selected randomly and represented as b_{selected} which is selected from the set R of remaining unused bugs:

$$b_{\text{selected}} = R(B_{\text{unused}})$$

We will be slicing the selected bug b_{selected} with the given f_{selected} into the fc. The slicing location is determined by the values within the source attribute, where the start and length values are represented in sequence while delimited by colons. Given that, slicing is done to place in the bug snippet and it is represented by the following function: Let f_c^* be the new file content that has the bug injected:

$$f_c^* = SliceIn(f_c, b_{\text{selected}}, f_{\text{selected}})$$

Finally, after each successful injection b_{selected} will be appended into B^*_{used}. The updated set B^*_{used} of used bugs can be obtained as:

$$B^*_{\text{used}} = B_{\text{used}} \cup \{b_{\text{selected}}\}$$

After each injection, B^*_{used} will be saved and kept for feature extraction use later on.

With the bug injection process explained, we now transition to discussing our feature extraction methodologies.

B. FEATURE EXTRACTION

Models will be built with features extracted from smart contracts that have been injected with the predefined bugs as mentioned in the previous section. However, there is one key thing to be noted, as each injection location is randomly selected we would not have the same set of code for the 7 categories of smart contract bugs. This will be further explained in each feature extraction algorithm section. The vulnerabilities to be injected are Reentrancy, Arithmetic, Unauthorized send, Transaction Origin, Timestamp

Algorithm 4 Extract Opcodes From Bugged Files

procedure SimplifyOpcode(f_c, B^*_{used})
  Initialize $\mathcal{L}_{\text{opcodes}} \leftarrow \emptyset$
  $C \leftarrow \text{SOLC}(f_c)$
  for $(ABI, O) \in C$ **do**
    found_match $\leftarrow$ False
    for $bug \in B^*_{\text{used}}$ **do**
      for $G_i \in ABI$ **do**
        if name$_i$ = bug.name **then**
          found_match $\leftarrow$ True
          $O^* \leftarrow \text{Simplify}(O)$
          $\mathcal{L}_{\text{opcodes}} \leftarrow \mathcal{L}_{\text{opcodes}} \cup \{O^*\}$
        end if
      end for
      if found_match **then**
        Break
      end if
    end for
  end for
  return $\mathcal{L}_{\text{opcodes}}$
end procedure

dependency, Transaction Order Dependency, and Unhandled Exceptions.

1) OPCODES

Opcodes are obtained by using SOLC by using the input and output JSON method — which is also the recommended way that has a consistent interface throughout all compiler versions. As not all contracts within each Solidity file will have a bug injected, a cross-check will be needed to only take in the opcode from injected contracts. Hence, given the file name and B^*_{used}, we will sift out contracts by comparing the function name with the *Abstract Binary Interface*(ABI). The full algorithm of the Opcode extraction process can be found in Algorithm (4).

As a next step, to describe the sifting process, let's consider C to be the set of m contracts returned by the solc. Each contract in C contains both an Application Binary Interface (ABI) and an ordered list of opcodes. Formally, C is defined as:

$$C = \{(ABI_1, O_1), (ABI_2, O_2), \ldots, (ABI_m, O_m)\}$$

where ABI_i represents the Application Binary Interface for the i^{th} contract, and O_i represents the ordered list of opcodes for the i^{th} contract. Also, $ABI_i = \{G_1, G_2, \ldots, G_r\}$, where each G_i is a function descriptor with a maximum number of r function descriptors per ABI.

Each function descriptor G_i could be represented as a tuple:

$$G_i = \big(\text{constant}_i, \text{inputs}_i, \text{name}_i, \text{outputs}_i, \text{payable}_i, \text{stateMutability}_i, \text{type}_i\big)$$

Each element in G_i would map to the corresponding property in the ABI JSON.

Following this, opcodes will be simplified. Simplification is done because opcodes such as PUSH have 32 variations while SWAP, and DUP have 16. Where each variation represents the number of bytes to be pushed on the stack. i.e., PUSH1 - 1 byte, PUSH4 - 4 bytes. The simplification rules enforced are similar to [19] and can be found in Table 3. With this simplification, the number of opcodes remaining will only be 77, thus reducing the dimension of the feature vector. We then employ the `n-gram` algorithm [32] for feature extraction. In natural language processing and computational linguistics `n-gram` is widely used as a calculation of the frequency distribution of a selected n-number of tokens, where n refers to the number of adjacent elements to consider from a string of tokens. Unigrams, bigrams, and trigrams are some examples of n-grams where n is 1,2 or 3, respectively [19]. For this paper, we have chosen the trigram approach for feature extraction. This choice is motivated by the need to capture more information while maintaining scalability, particularly as additional bugs are incorporated, allowing for more syntax to be captured for precise classification. Additionally, all hexadecimal values are removed, as they are unique to each smart contract and do not provide any flow information relevant to bug identification.

The vectorizing of text features is done by Term Frequency-Inverse Document Frequency (*Tfidf*). `Tfidf` can be broken down into two sections, I) Term Frequency, 2) Inverse Document Frequency. **1) Term frequency** will reflect frequently occurring sections of a document by using a weighting factor, in which the weight increases proportionally to the number of times a word appears in the document. However, this is offset by **2) Inverse Document Frequency** where it buries commonly appearing words and highlights rare occurring words. This allows unique features to surface.

Post extraction, we are left with a sparse matrix. To further reduce the dimension for model training, SelectKBest based on Chi-square and TruncatedSVD from scikit learn API [33] are both employed. SelectKBest will select the best 2000 features from the sparse matrix followed by the TruncatedSVD. Rather than the more commonly known dimension reduction method — Principle Component Analysis (PCA), which does not work on sparse matrix. TruncatedSVD is able to work on it efficiently because it does not center the data before computing the single value decomposition. The number of components are based on the cumulative variance of 95% from the selected 2000 features. With this method, we will obtain the features that have a cumulative variance of at least 95% and thereby reducing the number of features.

Upon completion of these steps, the Opcode trigram feature vector is primed and ready for the machine learning model training.

Algorithm 5 Extract CFG From Bugged Smart Contracts

procedure RuntimeBytecode(f_c, B^*_{used})
  Initialize $\mathcal{L}_{\text{cfg}} \leftarrow \emptyset$
  $C \leftarrow \text{SOLC}(f_c)$
  for $(ABI, BC) \in C$ **do**
    found_match $\leftarrow$ False
    for $bug \in B^*_{\text{used}}$ **do**
      for $G_i \in ABI$ **do**
        if name$_i$ = bug.name **then**
          found_match $\leftarrow$ True
          $CFG \leftarrow \text{Ethersolve}(BC)$
          $\mathcal{L}_{\text{cfg}} \leftarrow \mathcal{L}_{\text{cfg}} \cup \{CFG\}$
        end if
      end for
      if found_match **then**
        Break
      end if
    end for
  end for
  return $\mathcal{L}_{\text{cfg}}$
end procedure

C. CONTROL FLOW GRAPH

Other than extracting static features from opcodes using `n-gram`, constructing a CFG has more benefits as it contains sequence data from the runtime opcodes. Allowing much more intrinsic patterns to surface. In this work, we employ a CFG builder from Ethersolve [34]. It is a tool that uses symbolic stack execution to resolve jump destination, resulting in accurate edges. As Ethersolve is built using Java, we have utilised its' core module by creating a wrapper in python contained within a docker instance.

Similar to the process done in opcode for contract selection, we have done the same by obtaining deployed bytecode from solc and only process contracts that has bug function inside the ML model training. Full algorithm can be found in Algorithm 5. Below we will describe each representation.

Let C be the set of contracts returned by the solc. Each contract in C contains both an Application Binary Interface (ABI) and an ordered list of opcodes. Formally, C is defined as:

$$C = \{(ABI_1, BC_1), (ABI_2, BC_2), \ldots, (ABI_m, BC_m)\}$$

where, ABI_i represents the Application Binary Interface for the i^{th} contract, and BC_i represents the ordered list of deployed bytecode for the i^{th} contract.

Graph information was subsequently extracted using PecanPy [35], a fast, efficient, and parallelized Python implementation of Node2Vec [36]. Although Node2Vec is adept at learning low-dimensional representations of nodes within a graph, it lacks parallelization in its random walks. Addressing this limitation significantly enhances the efficiency of learning in dense networks [35].

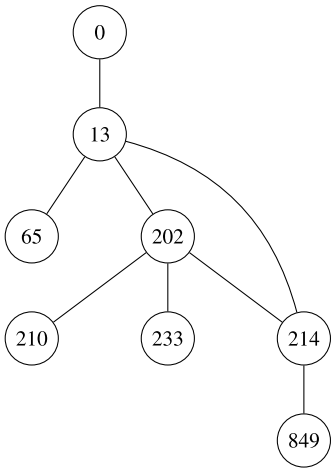


FIGURE 9. A snippet of digraph from Ethersolve CFG with parameters $p = 2, q = 0.5$.

During each walk, the gathered information is input into a Word2Vec model, resulting in node embeddings. For this purpose, the *Precomp* model is employed. In *Precomp*, the return parameter **p** is set to 2, and the in-out parameter **q** is 0.5. This model represents an optimized version of Node2Vec, precomputing and storing all transition probabilities for random walks. The values of p and q significantly influence the structure of the resultant graph. The return parameter p influences the random walk's likelihood of selecting an immediately preceding node, while the in-out parameter q encourages the walk to remain within its local neighborhood. With a lower value of q, the walk is more likely to visit nodes that are also connected to the previous node.

The final embedded information is constrained by available resources, therefore the number of walks from each node will be limited to 1, and walk length will be set to the average graph length across all categories, including *Clean* contracts. Walks shorter than this average are padded when the depth of the selected node is less than the average walk length.

Given the parameters and constraints, we can now do a small example on how Node2vec walks are done using a tree snippet generated by Ethersolve CFG at Figure 9. Probabilities of paths are grouped into 3 types, direct return node, node that does not have a path to the direct return node and node that has a return edge to the direct return node. Assuming that we will start at node 202, it has neighbours 13, 210, 233, 214 and the return node is 13. The transition probabilities for each paths will be $\frac{1}{p}$ to 13 as it is the return node, $\frac{1}{q}$ for node 210 and 233 as they do not have a return edge to node 13 and 1 for node 214 as it has a return path to node 13. It will then be normalized by the sum of each possible paths, which will give us probabilities of 0.111 for a walk going back to 13, 0.444 to node 210 and 233, finally 0.222 to node 214. Given as such, the next walk will most probably be either 210 or 233.

After obtaining the embedded data, a Long Short Term Memory (LSTM) model will be used for the evaluation.

TABLE 3. Opcode simplification.

Substituted Opcodes	Original Opcodes
ARITHMETIC_OP	ADD MUL SUB DIV SDIV SMOD
CONSTANT1	BLOCKHASH TIMESTAMP NUMBER DIFFICULTY GASLIMIT COINBASE
CONSTANT2	ADDRESS ORIGIN CALLER
COMPARISON	LT GT SLT SGT
LOGIC_OP	AND OR XOR NOT
DUP	DUP6-DUP16
SWAP	SWAP6 - SWAP16
PUSH	PUSH6 - PUSH32
LOG	LOG1-LOG4

Model training was implemented based on dataset size supportable by available computational resources.

On algorithmic complexity for the Pre-processing algorithm (Algorithm 1) and Bug insertion algorithm (Algorithm 3), the worst-case complexity depends on the size of the smart contract, $\mathcal{O}(\mathcal{S})$, where $\mathcal{S}$ represents the number of lines in the smart contract. Whereas the complexity of the Function filtering algorithm (Algorithm 2) is determined based on the number of nodes in the AST tree, $\mathcal{O}(\mathcal{N})$. In order to extract the opcodes from a given smart contract, the complexity depends on the total number of function descriptors in an ABI (r) and the total number of ordered lists of ABI, opcodes within a contract (m). Hence the algorithmic complexity for extracting opcodes from a smart contract (Algorithm 4) is in the order of polynomial complexity, $\mathcal{O}(rm^2)$. For a smaller number of opcodes within a contract (which is usually true in practical smart contract development), this time is much less. This can be observed from Figure 13. The algorithmic complexity for the feature selection algorithm using the `n-gram` algorithm is given by $\mathcal{O}(\mathtt{n}^2 \log \mathtt{n})$. Thus the worst-case complexity for Algorithm 5 that extracts the run-time bytecode is given by $\mathcal{O}(rm^2\mathtt{n}^2 \log \mathtt{n})$. For a small number of opcodes within a smart contract and $\mathtt{n} = 3$ (for the trigram feature selection algorithm), the polynomial complexity scales down to a practical running time for detecting smart contract vulnerabilities.

D. MACHINE LEARNING MODELS FOR CLASSIFICATION

Our dataset encompasses eight distinct categories, necessitating the use of a multiclass classifier algorithm for the training of classifiers. These algorithms are capable of handling multiple classes without the need to be adjusted or modified. In this research, we will be using the following models from scikit-learn api [33] for comparison: Random Forest (RF), XGBoost (XGB), Support Vector Machine (SVM) and Multi-layer Perceptron (MLP) which is a Neutral-Network implementation of scikit-learn.

We will also utilize sequential neutral network models such as Long Short-Term Memory (LSTM). LSTM is a variant of Recurrent Neutral Network (RNN) and it is designed to address the problem of vanishing gradients that arises during the back propagation process. The vanishing gradients is a problem because weights of edges are adjusted according to the calculated loss. This inhibit long sequences from updating earlier layers as the update value is according to the difference from the last layer. The main difference between LSTM and its predecessor is the ability to retain more information within each cell. This is done so by having 'gates' to control the amount of information flow in the network. However, after a preliminary study, CFG data with LSTM did not yield good results and we therefore decided to combine the MLP model into the process making it into a multi-model. MLP model is chosen because the training epoch can be shared between both models, allowing for simpler implementation. Inputs are the `Tfidf` and CFG data. This allows increased depth and complexity, resulting in better performance in terms of generalizing the features.

TABLE 4. Variables used in the MythSlith Algorithm.

Variable	Description
mythrilDepth	The depth of mythril symbolic execution.
Vulnerability	The current detection vulnerability
sc	Smart contract
M_r	Mythril scanning result
S_r	Slither scanning result
$DeepM_r$	Deeper depth Mythril scanning result

To ensure a well-trained model, we have implemented several measures such as $k-fold\ validation$, $learning-curve$ and $roc\ curve$. These checks are done to all models before passing off for the unseen data test. K-fold validation is a data partitioning method for assessing model performances. This method evaluate how well a model generalise an independent dataset. This is done so by having k folds, where k refers to a number of divided parts for the data. After dividing, 1 part of the k fold will be used as the independent test data while the rest would be the training set. This method provides variance and bias reduction ensuring a more accurate estimation. The purpose of learning curves is to identify 2 undesirable states during the training phase for the models, the 2 states are *Underfitting* and *Overfitting*. These states are identified by looking at it's training scores, since this is a classification task we will be using accuracy as the score. Underfitting occurs when both training and validation scores are low and continue to persist when more training data is added. Which infer that the model is too simple and is incapable of capturing the underlying pattern. Overfitting is when training score is high while the validation score is significantly lower. This infer that model may be too complex and will most likely fail at generalizing the different classes. Learning curves of each model can be found in Figure 11a. Epochs training information is much suited for neural network representation, for our MLP and MLP + LSTM model learning curve comparison can be found in Figure 11b.

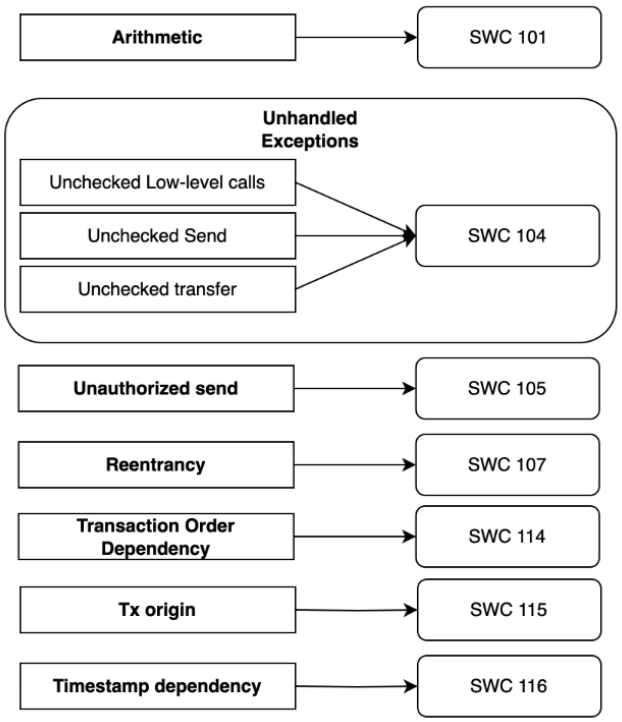


FIGURE 10. Mapping of Slither vulnerabilities to SWC.

The ROC curve analysis is utilized to evaluate each model's ability to distinguish specific vulnerabilities from the rest. These curves, shown in Figure 12, reveal that the RF and MLP + LSTM models struggle to effectively differentiate between the classes.

E. MULTI-MODEL APPROACH

Building upon traditional models, we have further developed a multi-model approach utilizing a combination of `Tfidf` vectorized opcodes and PecanPy node embeddings, employing a Keras multiple input model. This technique enables the use of multiple sub-networks, which can be concatenated or merged into a unified network at a certain point. This method proves particularly advantageous when dealing with diverse data sources, allowing for each to be processed by distinct models.

In the framework of this multi-model architecture, we employed an MLP for opcode data and an LSTM for PecanPy data processing. However, due to the extensive size of the dataset, we limited our test to only 50 randomly selected, bugged Solidity contracts from each category.

F. DESIGN OF MYTHSLITH

Analysis tools are considered the go-to when checking for any smart contracts vulnerabilities. However, contemporary tools are not yet capable of a full detection [37]. Hence in order to have a guideline for the ML model, Mythril and Slither is used.

Both Mythril and Slither are considered as static tools, however, Mythril uses symbolic execution and a SMT solver,

Algorithm 6 MythSlith Algorithm

```
mythrilDepth ← 22
if Vulnerability = Reentrancy then
    S_r ← Slither(sc)
    output S_r
else if (Vulnerability = Unauthorizedsend) or
    (Vulnerability = Txorigin) or
    (Vulnerability = Arithmetic) then
    M_r ← Mythril(sc, mythrilDepth)
    output M_r
else
    M_r ← Mythril(sc, mythrilDepth)
    if M_r.Severity = High then
        output M_r
    else
        mythrilDepth ← 100
        S_r ← Slither(sc)
        DeepM_r ← Mythril(sc, mythrilDepth)
        if DeepM_r.Severity = High then
            output DeepM_r
        else if S_r.Severity = High then
            output S_r
        else if (DeepM_r.Severity = Other) or
        (S_r.Severity = Other) then
            if DeepM_r.Severity = Other then
                output DeepM_r
            end if
            if S_r.Severity = Other and
|DeepM_r.MediumSeverity| < |S_r.MediumSeverity| and
|DeepM_r.LowSeverity| < |S_r.LowSeverity| then
                output S_r
            end if
        end if
    end if
end if
```

Z3, to determine the satisfiability of the generated symbolic formula, which is a mix of dynamic and formal verification. In [24] an empirical study on available open-sourced tools was done, the result for their curated dataset, which was collected from real vulnerable contracts or injected with bugs, showed that Mythril yields an accuracy of 27% while Slither 17%, though low but they are the highest among all the 9 tools presented. In conclusion, [24] suggested to use of a combination of Mythril and Slither which could yield a 37% accuracy.

Certainly, there are also other open-sourced tools available but, Mythril and Slither are both actively updated and therefore more suited for a direct comparison. We have designed a new integrated algorithm, MythSlith, a simple and elegant combination of both tools, which takes reference from a normal depth Mythril analysis to decide on the process. Mythril is used as the baseline for MythSlith because it has a higher detection accuracy as shown

in [24]. Furthermore, specific tools have been designated for addressing vulnerabilities like Reentrancy, Unauthorized send, Tx origin, and Arithmetic. This allocation stems from the outcomes detailed in Table 5. Additionally, although the severity level was initially absent in Slither, it has been incorporated, referencing the SWC registry and aligning with the identified vulnerabilities. The algorithm design can be found in the Algorithm 6. The variables used in MythSlith Algorithms is listed in Table 4.

G. CHALLENGES

1) MULTIPROCESSING

Multiprocessing is implemented to increase the time taken for test, however, this has spun off a new problem while changing solc version. Therefore, in order to prevent compilation error, all processes are dockerized and solc-select is included inside both Mythril and Slither dockerfile. Any action that has got to deal with solc will be spinning up a docker container to prevent any compiler mismatch issue.

2) UNIFIED VULNERABILITY CLASSIFICATION

Combining Mythril and Slither poses another challenge which is the definition of vulnerabilities. Though both tools are capable of detecting some common bugs, they do not use the same standard. While Mythril uses the more recognized SWC, Slither uses its own definition. To resolve this, we have added in SWC definition in the Slither code and incorporated the output with SWC ID. This will allow both tools to communicate with reference to the same vulnerabilities definition standard. In this paper, we have mapped the seven different vulnerabilities that can be found in Slither to SWC ID. The mapping can be found in Figure 10.

V. RESULTS AND PERFORMANCE ANALYSIS

In this section, experiment results from both analysis tools and our ML based approach will be put against each other for comparison. Effectiveness of each tool will be based on four parameters namely, (i) accuracy, (ii) precision, (iii) recall, and (iv) F1 score. These parameters will then form the confusion matrix for better visualization. All experiments were conducted on a machine with the following specifications: R162-ZA1-00 with 16 CPUs x AMD EPYC 7282 16-Core Processor 64GB of RAM and 1.9TB of SSD.

In this paper, a total of 4335 manually verified Solidity files were sourced from the smart contract sanctuary [31] repository to track verified smart contracts from the Ethereum mainnet and testnets, such as rinkeby, ropston, kovan, etc. And also from Binance Chain, Polygon/Matic, and Tron. In the mainnet repository, contract versions have a wide spread from 0.4.1 to 0.8.7 given the latest update. For this experiment, we will initially consider contracts from the mainnet and Solidity versions from 0.4.11 to 0.4.26 to compare against the known ground truths. Any contracts below 0.4.11 will not be considered due to a known compiler bug where source indexes could be inconsistent between different Solidity compiler version. Inconsistency between the source indexes will affect the bug injection process where it is reliant on it to insert bugs.

The unseen test set is prepared by a random selection of 100 solidity files from the Smart Contract Sanctuary. These files are in addition to the existing 4335 contracts. These contracts will then go through the same bug injection and feature extraction procedure as in Algorithm 3 and Algorithm 4, Algorithm 5.

Complete results can be found in both Table 6 and Table 5. Table 6 illustrates the overall comparison between analysis tools and ML models.

A. EVALUATION METRICS

To enable the comparison, SWC standard is utilized as the benchmark and it can be found in Fig 10. The results of the analysis tools and ML models are obtained by executing the same set of data with seven different classifications of vulnerabilities. The clean category is also added for clarity. To ensure a balanced and unbiased result from the analysis tool, each tool will run the clean dataset and then the result will be used as the baseline, ensuring a reference ground truth. This is because we can never assume each smart contract is free of bugs.

Given the result from the analysis, a confusion matrix will be constructed for the evaluation. The confusion matrix comprises the following values in accordance with the result prediction. These values with description can be found in Table 7.

With these values, we will be able to construct metrics:

- **Accuracy**: represents ratio of correctly predicted values to the total dataset. It is a measure of the overall correctness.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{1}$$

- **Precision**: can also be referred as positive predicted value, represents the number of correctly classified positive values to the total predicted positives.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

- **Recall**: also known as true positive rate, hit rate or sensitivity, represents the ratio of correctly classified instances to all datasets.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3}$$

- **F1 Score**: more useful for uneven dataset or class distribution. This metric is the weighted average for Precision and Recall.

$$\text{F1 Score} = \frac{2 \times (\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}} \tag{4}$$

- **False Positive Rate**: is a proportion for the measure of incorrectly classified instances as positive.

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{5}$$

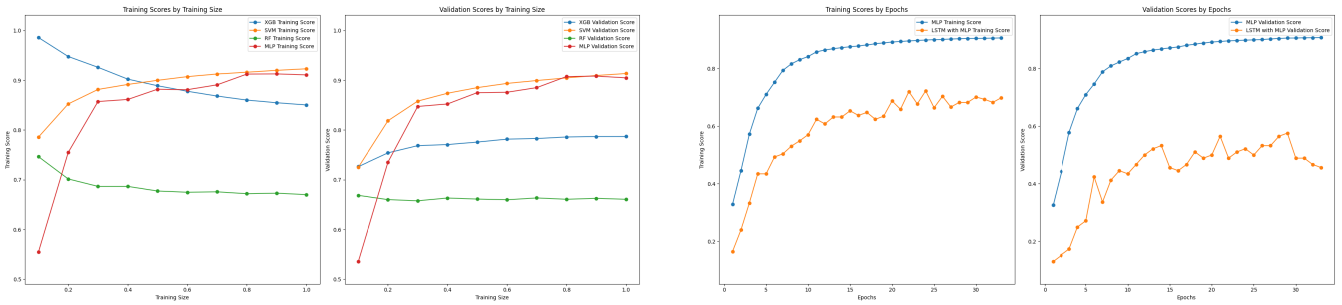
TABLE 5. Comparison of different tools.

	Tool	Precision	Recall	F1 Score	FPR
Clean	Mythril	0.1474	0.0955	0.1159	0.2590
	Slither	0.3750	0.2247	0.2810	0.0715
	MythSlith	0.1185	0.0323	0.0508	0.1045
	SVM	0.8289	0.8417	0.8352	0.0365
	RF	0.5699	0.6627	0.6128	0.0268
	XGB	0.8732	0.6911	0.7716	0.0211
	MLP	0.8880	0.8571	0.8723	0.0227
	MLP + LSTM	0.7692	0.1931	0.3086	0.0121
Reentrancy	Mythril	0.1889	0.7321	0.3004	0.1004
	Slither	0.4354	0.5120	0.4706	0.0557
	MythSlith	0.2656	0.5120	0.3497	0.1108
	SVM	0.7940	0.9518	0.8658	0.0309
	RF	0.5699	0.6627	0.6128	0.0626
	XGB	0.5738	0.8193	0.6749	0.0762
	MLP	0.8514	0.8976	0.8739	0.0196
	MLP + LSTM	0.3443	0.1265	0.1850	0.0301
Arithmetic	Mythril	0.2902	1.0000	0.4498	0.1043
	Slither	0.0000	0.0000	0.0000	0.0000
	MythSlith	0.2931	1.0000	0.4533	0.0992
	SVM	0.8710	0.8804	0.8757	0.0183
	RF	0.6606	0.5924	0.6246	0.0428
	XGB	0.8047	0.7391	0.7705	0.0252
	MLP	0.8967	0.8967	0.8967	0.0145
	MLP + LSTM	0.4815	0.0707	0.1232	0.0107
Tx origin	Mythril	0.9505	1.0000	0.9746	0.0062
	Slither	0.9800	0.4414	0.6087	0.0007
	MythSlith	0.9897	1.0000	0.9948	0.0013
	SVM	0.9639	0.9397	0.9517	0.0054
	RF	0.6260	0.7990	0.7020	0.0735
	XGB	0.7269	0.8693	0.7918	0.0502
	MLP	0.9552	0.9648	0.9600	0.0070
	MLP + LSTM	0.5759	0.7437	0.6491	0.0843
Unauthorized Send	Mythril	0.8571	1.0000	0.9231	0.0106
	Slither	0.3225	1.0000	0.4877	0.1373
	MythSlith	0.5192	1.0000	0.6835	0.0620
	SVM	0.8988	0.8483	0.8728	0.0129
	RF	0.6400	0.7191	0.6772	0.0548
	XGB	0.8075	0.7303	0.7670	0.0236
	MLP	0.9205	0.9101	0.9153	0.0106
	MLP + LSTM	0.9167	0.1236	0.2178	0.0015
Timestamp Dependency	Mythril	0.3171	0.2671	0.2900	0.0505
	Slither	0.7273	0.6667	0.6957	0.0179
	MythSlith	0.4174	0.3019	0.3504	0.0428
	SVM	0.9752	0.9691	0.9721	0.0030
	RF	0.6754	0.7963	0.7309	0.0466
	XGB	0.8344	0.8086	0.8213	0.0195
	MLP	0.9695	0.9815	0.9755	0.0037
	MLP + LSTM	0.6854	0.7531	0.7176	0.0421
Transaction Order Dependency	Mythril	0.0000	0.0000	0.0000	0.0000
	Slither	0.0000	0.0000	0.0000	0.0000
	MythSlith	0.0000	0.0000	0.0000	0.0000
	SVM	0.9620	0.9672	0.9646	0.0053
	RF	0.7857	0.8415	0.8127	0.0321
	XGB	0.8844	0.8361	0.8596	0.0152
	MLP	0.9674	0.9727	0.9700	0.0045
	MLP + LSTM	0.8182	0.6393	0.7178	0.0199
Unhandled Exceptions	Mythril	1.0000	0.7683	0.8690	0.0000
	Slither	0.7377	0.4737	0.5769	0.0105
	MythSlith	0.8056	0.5859	0.6784	0.0086
	SVM	0.9197	0.7826	0.8456	0.0082
	RF	0.5691	0.4348	0.4930	0.0398
	XGB	0.7105	0.6708	0.6901	0.0330
	MLP	0.8544	0.8385	0.8464	0.0173
	MLP + LSTM	0.1791	0.8199	0.2940	0.4545

In this study, we will be looking at recall followed by precision and f1 score then accuracy. This is because missing FN has much higher consequences than an increase in FP, as FN can lead to potential security breaches and

TABLE 6. Overall comparison different tools.

	Accuracy	Precision	Recall	F1 score	FPR
Mythril	0.7516	0.7092	0.3108	0.4023	0.0664
Slither	0.6035	0.6133	0.1753	0.2430	0.0367
MythSlith	0.7388	0.6745	0.3065	0.3883	0.0536
RF	0.6676	0.6757	0.6718	0.6627	0.0473
MLP	0.9129	0.9128	0.9129	0.9127	0.0125
XGB	0.7681	0.7837	0.7681	0.7702	0.0330
SVM	0.8954	0.9017	0.8976	0.8979	0.0151
MLP + LSTM	0.4189	0.5963	0.4337	0.4017	0.0819



(a) Learning rate for XGBoost, SVM, RF, MLP.

(b) Learning rate for MLP Vs MLP+LSTM.

FIGURE 11. Learning rate with increasing training size (a) and increasing epochs (b).

TABLE 7. Variables used in evaluation metrics calculation.

Metrics	Description
TP	True Positive. This is when an instance has been correctly classified as the intended class.
TN	True Negative. This represents instances where it has been correctly classified as negative.
FP	False Positive. This is also known as Type 1 error, is when incorrect positive classification when in fact it is a negative class.
FN	False Negative. This is also known as Type 2 error, is when a positive case has been classified as negative

financial losses. However, it is advised to take both Precision and F1 scores into consideration to ensure a balanced model.

B. PERFORMANCE OF MACHINE LEARNING MODELS

In this section, we illustrate the performance comparison between our ML models, Mythril, Slither, and MythSlith on the 100 bugged solidity files. The overall performance analysis of the tools, as summarized in Table 6, shows that the MLP and SVM models exhibited the highest levels of accuracy, precision, recall, and F1 score, with MLP achieving an accuracy of 0.9129 and an F1 score of 0.9127, and SVM showing a slightly lower accuracy of 0.8954 and an F1 score of 0.8979. FPR of the models has shed some light on the performance, that the MLP among the all other models has the lowest FPR at 0.0125 — which indicates its likeliness to flag out False positive is unlikely.

The detailed results of the individual vulnerability analysis are presented in Table 5. This analysis revealed that MLP has consistently demonstrated superior performance across most of the vulnerability detection, particularly for the detection of Clean and Unauthorized send with f1 scores of 0.8723 and 0.9153 respectively. SVM did well across most of the categories, however, it has a little trouble with Reentrancy, while having a high recall of 0.9518 its' precision took a toll at 0.7940. The XGB model displayed a consistent performance across various categories, with a notable F1 score of 0.7716 in the Clean category, though it did not outperform all other tools in this category. Surprisingly, bagging ensemble learning technique like RF did not do very well, measures across the board is not more than 0.67. The multi-model of MLP and LSTM weak results, while doing well in Transaction order with F1 score of 0.7176, it is less effective in other categories. One notable FPR measure is for the multi-model, where it is 0.4545 for Unhandled Exceptions, this clearly indicates the low effectiveness of the features used for the models. While MLP excels in

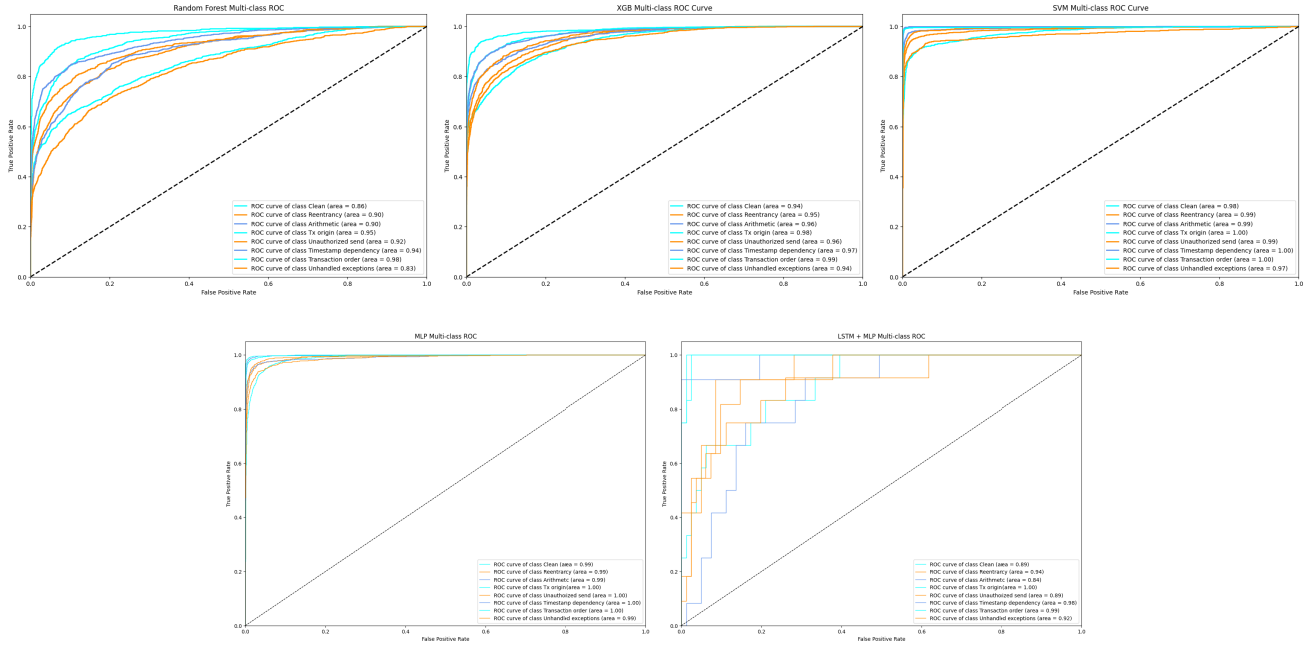


FIGURE 12. ROC Curves for various models. From left to right, top to bottom: Random Forest, XGBoost, SVM, MLP, MLP+LSTM.

Timestamp Dependency with the lowest measure at 0.0037, indicating high confidence in the detection.

C. PERFORMANCE COMPARISON WITH SOFTWARE VERIFICATION AND VALIDATION TOOLS

Results of the tools can be found at Table 5. Right off the bat, it is clear that current analysis tool is unable to identify some vulnerabilities. *Transaction order* seem to be a challenge for the analysis tools, while *Arithmetic* proof to be too hard for Slither to handle. In addition, while it seem great from the precision point of view for the 3 tools, recall and F1 score of the does not do very well. This has yet again emphasised on the findings from [17]. And current tools has difficulty identifying *Clean* contracts, with *Slither* leading the pack at only 0.375 in precision while the rest is well below 0.2. Another vulnerability to highlight is the *TimestampDependency*, surprisingly all 3 tools did not fare well with recall being lesser than 0.2. FPR of clean contracts for Mythril is particularly high at 0.2590, and this behaviour continues in Reentrancy and Arithmetic. However, it did really well in Unhandled exceptions where no FP were raised. Slither did really well in Tx origin with just 0.0007 for its FPR. However, such performance is not backed by its recall and f1 score which has clearly indicate poor TP. MythSlith results were not spectacular, it is just hovering between Mythil and Slither measures. One example is Timestamp Dependency where MythSlith has a FPR of 0.0428, Mythril and Slither have 0.0505 and 0.0179 respectively. Due to its design, it can never be better than the highest measure by either tools.

These findings highlight the varying strengths and limitations of each model or tool, underscoring the influence of vulnerability type on the effectiveness of detection methods.

VI. ANALYSIS AND INFERENCES

A. PRE-PROCESSING AND FEATURE EXTRACTION

In our methodology, opcode sequences are utilized and simplified technique aimed at enhancing variance. We then employ `Tfidf` technique with trigram-based feature extraction. While this approach is efficient and straightforward, it suffers from a lack of context awareness. This limitation stems from its focus on only three consecutive opcodes at a time, leading to a sparsity issue. The sparsity results from the model's limited exposure to examples, potentially compromising its accuracy and robustness.

Furthermore, the sparse nature of the `Tfidf` representation presents computational challenges, particularly as data volumes escalate. In the context of our opcode dataset, this method generates 4, 824 feature columns via the `Tfidf` vectorizer. Additionally, the process of opcode simplification, while conserving computational resources, introduces significant drawbacks. A primary concern is the discarding of hexadecimal values, resulting in the loss of vital address information crucial for source location tracing.

Looking ahead, the adoption of an alternative vectorizer to `Tfidf` could potentially yield improved contextual understanding, enhancing the models' learning capabilities.

B. COMPARISON OF MYTHSLITH AND MLP MODEL

From the test conducted, it is clear that relying on one tool for smart contract analysis is not ideal. Current Software and verification tools such as Mythril [14] and Slither [13] tend to be on the safe side because the result has high precision with low recall and f1 score. In our effort to combine them and weave through the cracks by constructing MythSlith, such behavior still exists. No significant detection progress was

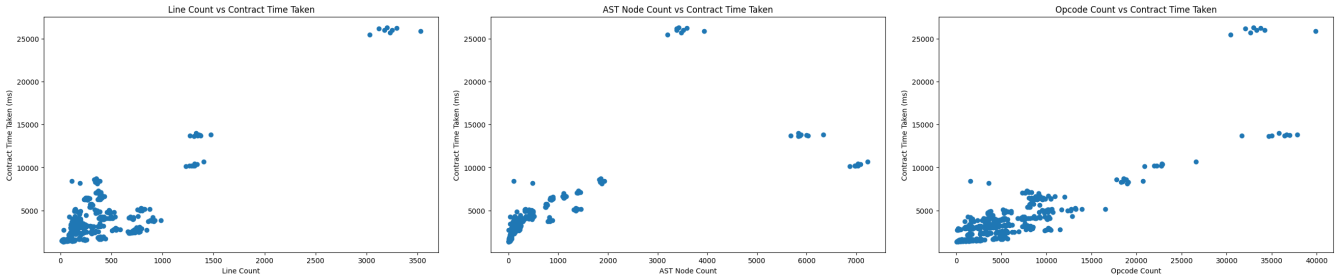


FIGURE 13. Pre-processing time (in *ms*) for opcode extraction with respect to (from left to right): Line Count, AST Node Count, and Opcode count, respectively.

TABLE 8. Average time taken for Smart contract vulnerability detection tool to analyze each smart contract.

Software Validation Tool	Average Time per Smart Contract (seconds)	Depth
Slither	5.36	-
Mythril	1270.33	22
MythSlith	1102.14	0/22/100
RF	4.45	-
MLP	4.43	-
XGB	4.20	-
SVM	6.46	-
MLP+LSTM	51.52	-

made, which then again is expected as no improvement was done to each tool. However, MythSlith did cover Slither's inability to detect Arithmetic vulnerabilities. This can be observed in Table 5.

In our proposed method, we show that detection via extracted features from smart contracts presents viable patterns for machine learning models to learn and classify them. However, the suitability of different models varies for this specific application, this can be clearly seen from the results of RF. Unlike the gradient correction method employed by XGB, the ensemble bagging approach of RF did not yield equally effective results. In terms of neural network method, MLP did well for all categories. In contrast, the multi-model of MLP + LSTM did not. This is primarily due to the CFG features derived from Ethersolve [34]. These CFG features did not present a strong pattern for our model to learn effectively. Furthermore, the generation of features using PecanPy [35] takes up a substantial amount of time due to random walks generation. However, the limited effectiveness of Ethersolve features in this context does not inherently diminish their value; the challenge may lie more with PecanPy's processing demands.

C. RUNNING TIME

In Table 8, we analyze and compare the running times for the various software validation tools. The running time for MythSlith is comparable to that of Mythril (or sometimes lesser). The average running time for MythSlith is 1102.14 seconds whereas that of Mythril is 1270.33 seconds. Note that unlike Mythril, the average running time of Slither is only 5.36 seconds since it does not involve the depth of the symbolic execution. The average running time for MythSlith is slightly less than that of Mythril since the depth of the symbolic execution is only increased for certain types of vulnerabilities, whereas for all other cases where Slither has better detection accuracy, MythSlith chooses Slither.

As observed in the Table 8, the running time to analyze a smart contract using ML models is much lesser (in the order of $<$ 7 seconds) compared to the software verification tools except for the multi-model MLP+LSTM since it involves two ML models. It is evident that the time taken to analyze the smart contract features and predict using the ML-model is real-time for the smart contract analysis. Whereas the pre-processing and training time for the ML-model is an offline process. The pre-processing time for opcode extraction with respect to the number of lines in the smart contract, number of AST nodes and number of CFG opcodes is depicted in Figure 13 from left to right, respectively. The maximum pre-processing time observed for a smart contract with 3500 lines is 26 seconds. While most practical smart contracts with less than 1000 lines of code take $<$ 10 seconds for pre-processing and opcode extraction.

VII. CONCLUSION AND FUTURE WORKS

Securing smart contracts is no easy feat, as they are not protected and visible to everyone. Current software verification tools tend to be on a defensive stance, only flagging the bugs if they are fully sure about it. This however would let the False negatives slip through. Therefore, we proposed a machine learning approach to effectively and efficiently detect seven types of vulnerabilities while also identifying clean contracts.

This was done by employing models such as Random Forest, XGBoost, Support Vector Machine, Multi-Layer Perception, and Long-Short Term Memory. To insert practical bugs and increase the vulnerability space, we proposed a practical bug injection technique that injects bugs into verified smart contracts that were cleaned using our proposed pre-processing algorithm. This helps to scale up the smart contracts' vulnerability space using features such as opcodes and CFG that were extracted for the model training. Prior

to vectorization, simplification was done to the opcodes in order to reduce the dimension and remove contract-specific hexadecimal values. TDIDF was utilized with trigram for the vectorization and the CFG data was processed by PencanPy where random walks are generated and vectorized with the Word2Vec model.

The results of the models were then benchmarked with software verification tools such as Mythril, Slither, and an experimental tool proposed in our work — MythSlith. From the results, machine learning models have shown superior performance in vulnerability detection than existing Software verification tools. While testing with real-time smart contracts, the MLP model performs the best at having 91% accuracy along with higher recall and f1-scores.

The FPR measures show that MLP achieved the best performance with the lowest average among all other tools at 0.0125, while MLP+LSTM achieved the lowest FPR of 0.0015 for unauthorized send.

While the current model improves the accuracy with significantly less FPR while detecting contract-wide bugs, it can further be improved by pinpointing the exact source location. One possible solution is to use a combination of fuzzing with formal verification to extend our current model to add this feature. This solution however will face a constraint that model patterns cannot be traced back to the source location. We can explore some viable solutions for this constraint by tagging the source index (that may not be used for model training) to a specific feature, thereby allowing traceback to the source. Bug injection method ensures that newer forms of features and inherent patterns are added. The current bug injection method leverages on function descriptors. However, not all bugs are in the form of functions. In future works, we will explore more generic bug injection methods.

REFERENCES

[1] PR Newswire. (2023). *Global Smart Contracts Market to Reach USD 9850 Million by 2030 With 24% CAGR | Revolutionizing Contract Management, Exploring the Opportunities and Trends Report By Zion Market Research*. Accessed: Sep. 3, 2023. [Online]. Available: https://finance.yahoo.com/news/global-smart-contracts-market-reach-160000824.html

[2] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on Ethereum smart contract vulnerabilities: A survey," 2019, *arXiv:1908.08605*.

[3] K. Ramana, R. M. Mohana, C. K. Kumar Reddy, G. Srivastava, and T. R. Gadekallu, "A blockchain-based data-sharing framework for cloud based Internet of Things systems with efficient smart contracts," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, May 2023, pp. 452–457.

[4] W. Wang, Z. Han, T. R. Gadekallu, S. Raza, J. Tanveer, and C. Su, "Lightweight blockchain-enhanced mutual authentication protocol for uavs," *IEEE Internet Things J.*, early access, Oct. 13, 2023, doi: 10.1109/JIOT.2023.3324543.

[5] J. Korn. (Aug. 2022). *Report: $1.9 Billion Stolen in Crypto Hacks So Far This Year*. Accessed: Aug. 16, 2022. [Online]. Available: https://edition.cnn.com/2022/08/16/tech/crypto-hack-rise-2022/index.html

[6] Binance. (2023). *Poolz Finance Hacked, Token Price Drops 93%*. Accessed: May 23, 2023. [Online]. Available: https://www.binance.com/en/feed/post/309330

[7] SolidityScan. (2023). *Poolz Finance Hack Analysis: Still Experiencing Overflow*. SolidityScan Blog. Accessed: May 23, 2023. [Online]. Available: https://blog.solidityscan.com/poolz-finance-hack-analysis-still-experiencing-overflow-fcf35ab8a6c5

[8] OpenZeppelin. (2022). *Developing Smart Contracts*. Accessed: Dec. 26, 2022. [Online]. Available: https://docs.openzeppelin.com/learn/developing-smart-contracts

[9] ConsenSys. (2022). *Smart Contract Best Practices*. Accessed: Dec. 26, 2022. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/

[10] Crytic. (2022). *Detector Documentation*. Accessed: Dec. 26, 2022. [Online]. Available: https://github.com/crytic/slither/wiki/Detector-Documentation

[11] SmartContractSecurity. (2022). *Smart Contract Weakness Classification Registry*. Accessed: Dec. 26, 2022. [Online]. Available: https://swcregistry.io/

[12] NCC Group. (2024). *Top 10 Decentralized Application Security Risks*. Accessed: Dec. 26, 2022. [Online]. Available: https://dasp.co/

[13] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.

[14] B. Mueller, "Smashing Ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, vol. 9, p. 54, Apr. 2018.

[15] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 1186–1189.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[17] X. Tang, K. Zhou, J. Cheng, H. Li, and Y. Yuan, "The vulnerabilities in smart contracts: A survey," in *Proc. 7th Int. Conf. Adv. Artif. Intell. Secur. (ICAIS)* 2021, Dublin, Ireland. Cham, Switzerland: Springer, Jul. 2021, pp. 177–190.

[18] P. Momeni, Y. Wang, and R. Samavi, "Machine learning model for smart contracts security analysis," in *Proc. 17th Int. Conf. Privacy, Secur. Trust (PST)*, Aug. 2019, pp. 1–6.

[19] 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, no. 2, pp. 1133–1144, Apr. 2021.

[20] S. Shakya, A. Mukherjee, R. Halder, A. Maiti, and A. Chaturvedi, "SmartMixModel: Machine learning-based vulnerability detection of solidity smart contracts," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Aug. 2022, pp. 37–44.

[21] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 415–427.

[22] T. M. Corporation. *Common Weakness Enumeration*. Accessed: Dec. 28, 2022. [Online]. Available: https://cwe.mitre.org/

[23] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.

[24] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Montreal, QC, Canada, Oct. 2020, pp. 530–541.

[25] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H. Lee, "Systematic review of security vulnerabilities in Ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.

[26] L. Duan, L. Yang, C. Liu, W. Ni, and W. Wang, "A new smart contract anomaly detection method by fusing opcode and source code features for blockchain services," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 4, pp. 4354–4368, Dec. 2023.

[27] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006.

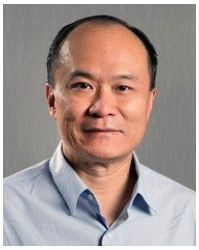
[28] T. Ball, "The concept of dynamic analysis," *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 216–234, Nov. 1999.

[29] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, and G. Tamburrelli, "Formal verification with confidence intervals to establish quality of service properties of software systems," *IEEE Trans. Rel.*, vol. 65, no. 1, pp. 107–125, Mar. 2016.

[30] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022.
[31] M. Ortner and S. Eskandari. *Smart Contract Sanctuary*. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary
[32] W. B. Cavnar and J. M. Trenkle, "*N*-gram-based text categorization," in *Proc. 3rd Annu. Symp. Document Anal. Inf. Retr. (SDAIR)*, vol. 161175. Las Vegas, NV, USA, 1994, pp. 1–14.
[33] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: Experiences from the scikit-learn project," in *Proc. ECML PKDD Workshop Lang. Data Mining Mach. Learn.*, 2013, pp. 108–122.
[34] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "EtherSolve: Computing an accurate control-flow graph from Ethereum bytecode," in *Proc. 29th IEEE/ACM Int. Conf. Program Comprehension*, May 2021, pp. 127–137.
[35] R. Liu and A. Krishnan, "PecanPy: A fast, efficient and parallelized Python implementation of *node2vec*," *Bioinformatics*, vol. 37, no. 19, pp. 3377–3379, Oct. 2021.
[36] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016.
[37] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey," 2022, *arXiv:2209.05872*.

LEE SONG HAW COLIN (Member, IEEE) received the bachelor's degree in mechatronics engineering from the Singapore Institute of Technology–University of Glasgow, in 2016. He is currently pursuing the Master of Engineering degree in future communications and blockchain with the Singapore Institute of Technology. He is a Research Engineer with the Singapore Institute of Technology. His professional journey includes a significant role in the development of a COVID-19 CMS application with the GP Connect Team. His current research interests include the intersection of AI, blockchain, and their applications in 5G networks.

PURNIMA MURALI MOHAN (Member, IEEE) received the M.S. and Ph.D. degrees in electrical and computer engineering from the National University of Singapore, in 2014 and 2018, respectively. She held a postdoctoral researcher position with the National University of Singapore, until 2018. She is currently an Assistant Professor with the Information and Communications Technology Cluster, Singapore Institute of Technology. She has expertise in Layer 2 and Layer 3 network protocols while working with the industry. Her current research interests include blockchain and AI, security in next-generation networks, optimization, and heuristics algorithm design.



JONATHAN PAN (Member, IEEE) received the Ph.D. degree in information technology and cyber security from Murdoch University, Australia. He is currently the Chief of the Disruptive Technologies Office and the Director of Cybersecurity of the Home Team Science and Technology Agency, which is a statutory board formed under Singapore's Ministry of Home Affairs to develop science and technology capabilities for the Home Team. He is also an Adjunct Associate Professor with Nanyang Technological University, Singapore. His research interests include cybersecurity, AI, and blockchain.

PETER LOH KOK KEONG (Senior Member, IEEE) received the M.Sc. degree in computer science from the University of Manchester, U.K., and the Ph.D. degree in computer engineering from Nanyang Technological University, Singapore. He is currently an Associate Professor with the Information and Communications Technology Cluster, Singapore Institute of Technology. He is a registered Professional Engineer in Singapore and a Chartered Engineer in U.K. He has more than 35 years of professional engineering, research, academic, and consultative experience. To date, he has authored/coauthored more than 100 publications, with several in high-impact, international, and peer-reviewed journals. His research interests include information and cyber security, data analytics and machine learning for digital crime, blockchain and the IoT, and malware analysis and classification.