SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing

Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, Chin-Wei Tien
Cybersecurity Technology Institute, Institute for Information Industry, Taipei Taiwan, R.O.C
Email: jianweiliao@iii.org.tw, tsungtatsai@iii.org.tw, ckhe@iii.org.tw, jakarence@iii.org.tw

***Abstract*—Blockchain has flourished in recent years. As a decentralized system architecture, smart contracts give the blockchain a user-defined logical concept. The smart contract is an executable program that can be used for automatic transactions on the Ethereum blockchain. In 2016, the DAO attack resulted in the theft of 60M USD due to unsafe smart contracts. Smart contracts are vulnerable to hacking because they are difficult to patch and there is a lack of assessment standards for ensuring their quality. Hackers can exploit the vulnerabilities in smart contracts when they have been published on Ethereum. Thus, this study presents SoliAudit (Solidity Audit), which uses machine learning and fuzz testing for smart contract vulnerability assessment. SoliAudit employs machine learning technology using Solidity machine code as learning features to verify 13 kinds of vulnerabilities, which have been listed as Top 10 threats by an open security organization. We also created a gray-box fuzz testing mechanism, which consists of a fuzzer contract and a simulated blockchain environment for on-line transaction verification. Different from previous research systems, SoliAudit can detect vulnerabilities without expert knowledge or predefined patterns. We subjected SoliAudit to real-world evaluation by using near 18k smart contracts from the Ethereum blockchain and Capture-the-Flag samples. The results show that the accuracy of SoliAudit can reach to 90% and the fuzzing can help identify potential weaknesses, including reentrancy and arithmetic overflow problems.**

Keywords—Smart contract, vulnerability, fuzz testing, machine learning

I. INTRODUCTION

Smart contracts have become the targets of hackers. In 2016, the DAO [1] attack led to the theft of approximately 60 million USD due to unsafe smart contracts, and hacking incidents targeting smart contracts continue to occur. In fact, instances of smart contract hacking have increased in frequency in recent years. After 2017, the interval between incidents has gradually shortened [2] .

Why have smart contracts become hacking targets? Their vulnerabilities are summarized in the following three points:

- Smart contracts are valuable but vulnerable.
- Because it is difficult to modify the state of the blockchain, smart contracts are difficult to patch.
- There is a lack of assessment standards for ensuring smart contract quality.

Like the OWASP TOP 10 [25], the NCC Group organization [7] proposed the Decentralized Application Security Project (DASP) TOP 10 [8]. The project lists the top 10 smart contract vulnerabilities, which are listed in TABLE I. Ranked from one to nine with a vulnerability index, each category has a different pattern. However, for the last category, "unknown unknowns," the pattern is still unidentified because smart contracts are still in their infancy.

TABLE I. DASP TOP 10

Index	*Category*	*Description*
1	Reentrancy	External contract calls are allowed to make new calls to the calling contract before the initial execution is complete.
2	Access Control	Insecure visibility settings give attackers straightforward ways to access a contract's private values or logic.
3	Arithmetic	Also known as integer overflow and integer underflow.
4	Unchecked Low-Level Calls	Unexpected behavior if return values are not handled properly.
5	Denial of Services	1. Maliciously behaving when acting as the recipient of a transaction. 2. Artificially increasing the gas necessary to compute a function. 3. Abusing access controls to access private components of smart contracts.
6	Bad Randomness	Sources of randomness are to some extent predictable, and malicious users can generally replicate it and attack the function by relying on its predictability.
7	Front Running	A malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution.
8	Time Manipulation	If a miner holds a stake on a contract, they could gain an advantage by choosing a suitable timestamp for a block they are mining.
9	Short Addresses	Short address attacks are a side effect of the Ethereum Virtual Machine (EVM) itself accepting incorrectly padded arguments.
10	Unknown unknowns	Unknown.

978-1-7281-2949-5/19/$31.00 ©2019 IEEE

In this research, we focus on Ethereum smart contracts, which are smart contracts written as code and compiled as a program to be deployed on the blockchain. As a decentralized system architecture, smart contracts give the blockchain a user-defined logical concept. Users can use smart contracts to create a transaction anonymously, and the execution results of smart contracts are stored in the blockchain and thus are difficult to delete or modify.

We propose a method that can analyze and detect vulnerabilities with machine learning, which only requires the sufficient samples to train and classify without requiring predefined or expert knowledge. Moreover, to deal with possible unknown vulnerabilities, we cannot only depend on compliance patterns. Therefore, we propose a dynamic fuzzer that can fuzz not only function parameters but also some parameters of the blockchain environment, like gas and ether. Our proposed method, called SoliAudit, uses static and dynamic testing technology via machine learning and a dynamic fuzzer to strengthen its contract vulnerability detection capabilities. The contributions of SoliAudit are summarized in the following three points:

- Designed a machine learning classification based on the smart contract opcode feature and achieved a vulnerability identification accuracy of up to 90% on 17,979 samples.
- Automatically generates a fuzzer contract for fuzz testing, and with abnormal analysis, successfully analyzed real world and CTF cases.
- Without expert knowledge and pre-defined features, our approach can rapidly adapt to new unknown weaknesses.

II. Related works

In this section, we compare three types of works or tools related to Ethereum smart contract vulnerability detection. They can be divided into three categories: compliance pattern analysis, symbolic execution, and fuzzer.

A. Compliance pattern analysis

The first tool is static analysis. Tsankov et al. proposed a method called Security [9] that uses a compliance pattern to check the vulnerability of a contract. Because the method requires expert knowledge to generate predefined patterns of vulnerability, it cannot detect unknown vulnerabilities.

B. Symbolic execution

The second category [10] uses symbolic execution to find vulnerabilities in smart contracts. However, these tools only use symbolic execution to check for the transformation of the memory state. The detection of vulnerabilities, like the static method which needed a compliance pattern, will eventually require expert knowledge. Also, this type of method executes the byte code and solves the constraints, and thus it is highly time consuming.

C. Fuzzer

Use of fuzzer in smart contract security is a new research topic recently. There are not many related work of fuzzer on smart contract in recent years. This method [13] runs the smart contract on the blockchain and fuzzes some input to the contract. However, it is like symbolic execution in that the fuzz testing result is part of a pattern for static analysis. Furthermore, like symbolic execution methods, it is also time consuming to run the test

As summarized above, current research methods for smart contract security require predefined patterns of vulnerability, and thus they cannot handle unknown vulnerabilities. Although some research uses dynamic testing, expert knowledge is still necessary for detecting vulnerabilities. Thus, we have proposed a method called SoliAudit that does not require expert knowledge or patterns. SoliAudit uses static machine learning with opcode and full dynamic fuzz testing with a running blockchain to detect vulnerabilities. Details on the method are introduced in the next section.

III. Proposed Architecture

In this section, we first introduce an overview of SoliAudit. Then, the detailed architectures of the vulnerability analyzer and dynamic fuzzer are introduced in subsections B and C, respectively.

A. Overview of SoliAudit

To detect vulnerabilities in Ethereum smart contracts, we proposed an architecture combining static and dynamic technologies via machine learning and a dynamic fuzzer to strengthen the method's contract vulnerability detection capabilities.

Fig. 1 shows an overview of the SoliAudit architecture, which contains two components: a vulnerability analyzer and a dynamic fuzzer. The brief input is an Ethereum smart contract source code. Each contract is compiled to opcode and verified by the vulnerability analyzer with a static machine learning classifier. Then, the contract is deployed and run on the Ethereum blockchain and is fuzzed by the dynamic fuzzer to find the vulnerability which was loosed or unknown with the vulnerability analyzer.

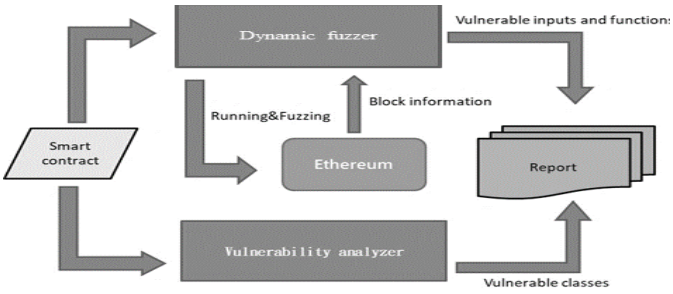

Fig. 1. Overview of SoliAudit architecture.

B. Vulnerability analyzer

Like most machine learning projects, our vulnerability analyzer [28] has two phases: one for training a predictable

model and the other for predicting vulnerabilities using the model. The architecture of the vulnerability analyzer is shown in Fig. 2.

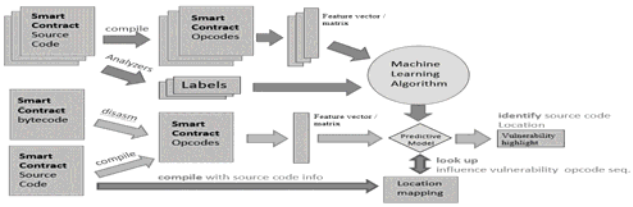

Fig. 2. Architecture of the vulnerability analyzer.

1) Training models:

a) Smart Contract Dataset: 21,044 smart contracts published on etherscan.io for Ethereum mainnet were collected. Duplicated contracts were filtered out according to their opcode sequences, and only the contracts that could be validly labeled were kept. Finally, 17,979 smart contracts [23] were used as the experimental dataset.

b) Labeling: We used Oyente[10] and Remix [26] to label our dataset with 13 vulnerabilities. Both tools provide command line execution and thus are able to label massive sets of smart contracts. Oyente performs analyses using symbolic execution and Remix using static analysis. TABLE II. lists all vulnerabilities that Oyente and Remix can detect.

c) Data Preprocessing: First, the smart contract sample is transformed to the opcode sequence, which can be derived from either the bytecode or the source code. Because smart contracts are stored in bytecode form on Ethereum mainnet but are usually published in source code form, using the opcode sequence as the analysis data makes our analyzer more flexible for use in a real-world environment. Then, the opcodes are processed by stemming. For example, opcode PUSH1, PUSH2, …, and PUSH32 were all stripped to PUSH. Thus, the stemming reduced 138 opcodes to 72 opcodes. Finally, we regarded some usual opcodes, like AND, OR, and LT, as stop words, and removed them. Finally, 30 ethereum-specific opcodes [23] remained and were used for feature extraction.

d) Feature Extraction: We attempted two different methods to extract features from the preprocessed opcode sequence:

i) *n-gram with term frequency–inverse document frequency (tf-idf)* [27]: This method splits each opcode sequence into n-gram tokens and represents each token as its tf-idf value. Ultimately, the feature for a sample is represented by a vector with dimensions equal to the number of possible token conbinations, and each dimension is mapped to the specified token and its tf-idf value in the dataset. An illustration of 2-gram with tf-idf is shown in Fig. 3, in which each dimension of the vector is the tf-idf value of 2-gram words.

ii) *Vectors trained by word2vec* [21]: Word2vec creates a word embedding that trains tokens, e.g. opcodes, as vectors in the specified space and attempts to maintain the semantic relations among vectors. Thus, in our work, each opcode is represented by a vector obtained by word2vec. Then, an opcode sequence is viewed as a list of vectors, and these vectors are concatenated row-by-row to form the feature matrix. An illustration is shown in Fig. 4, in which each opcode is replaced by a vector trained by word2vec and concatenated in the order of the opcode sequence to form the feature matrix.

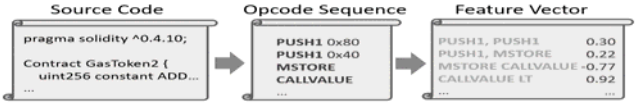

Fig. 3. An Illustration of feature extraction by n-gram with tf-idf.

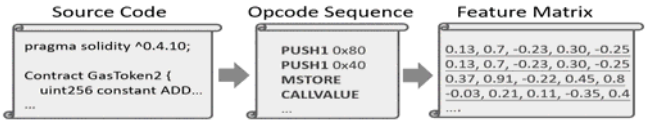

Fig. 4. An Illustration of feature extraction by word2vec.

e) Training: We used different training models for different features. For the n-gram with tf-idf feature vector, we trained the dataset using tranditional maching learning models, like Logistic Regression, SVM, Decision Tree, Random Forest, and K-Nearest Neighbor. Each vulnerability was trained independantly as a binary classification training. As for the word2vec-based feature matrix, the dataset was trained using a convolutional neural network (CNN) because the feature matrix preserves the ordering of the opcode sequence and CNN trains features considering the inner structure. In this situation, all vulnerabilities were trained simultaneously as a multi-label classification training.

TABLE II. VULNERABILITIES LABELED BY ANALYSIS TOOLS

Tools	*Vulnerabilities*	*Descriptions*	*DASP TOP 10 Category*	*Positive Samples*
Oyente	**Underflow**	Integer underflow	Arithmetic	11134
	Overflow	Integer overflow	Arithmetic	16012
	CallDepth	Use send or call cmd, but do not check the cmd result	Unchecked Return Values For Low Level Calls	407
	TOD	State will depend on the tx order	Front-Running	2594
	TimeDep	State will depend on the timestamp	Time manipulation, Bad Randomness	1228
	Reentrancy	Contract contains reentrancy function	Reentrancy	555
Remix	**AssertFail**	Contract contains the condition of assert fail	Denial of Service	7721
	TxOrigin	Contract use tx.origin	Access Control	155

CheckEffects	Contract checks if the state has been updated before the transaction or not	Unchecked Return Values For Low Level Calls	7183
InlineAssembly	Contract uses assembly code	Unchecked Return Values For Low Level Calls	1311
BlockTmestamp	Contract uses block.timestamp	Time manipulation, Bad Randomness	5879
LowlevelCalls	Contract uses send or call not transfer	Unchecked Return Values For Low Level Calls	5431
SelfDestruct	Contract uses selfdestruct	Denial of Service	1275

f) Predictive Models: After training, we acquired the predictive models for different machine learning algorithms. The experimental results of these models are shown in TABLE III. Between the predictive models, Logistic Regression scored the highest in our experiments.

2) Predicting Vulnerabilities: The next phase was to predict whether a test smart contract contained possible vulnerabilities using the predictive models obtained in the last step of the training phase.

a) Test Sample: Smart contract source codes or bytecodes were both acceptable. Both forms were transformed to their opcode sequences.

b) Feature Extraction: This step was identical to the feature extraction in the training phase. According to the predictive models, the feature could be a vector calculated from n-gram with tf-idf or a matrix derived from word2vec.

c) Predicting: It was predicted whether the test smart contract contained possible vulnerabilities. If the model was capable of calculating the most contributing n-gram tokens, i.e., n-gram opcodes, the opcodes for vulnerabilities that were predicted positive were also printed out.

d) Source Code Location Mapping: For the vulnerabilities predicted as positive, we attempted to map the vulnerable location in the source code. When the source code was provided, it would be compiled to a list of opcodes wherein each opcode had debug information of its corresponding location within the source code, e.g. line number and position in line. Thus, when the the most contributing n-gram opcode sequence was identified by models, this sequence would be used to match the list of opcodes and acquired the location within the source code.

C. Dynamic Fuzzer

A schematic of the dynamic fuzzer fuzz procedure is shown in Fig. 5. When the fuzzer acquires a test target contract, it generates a fuzzer contract and inserts an event to the target contract. The fuzzer fuzzes the target contract by fuzzer contract or by itself with different role accounts. After the fuzz ends, the fuzzer retrieves the block information for the event anomaly analysis and generates a report.

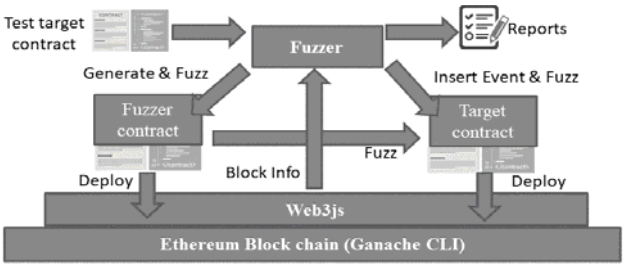

Fig. 5. Schematic of the fuzz procedure.

Fig. 6 shows the detailed architecture of the dynamic fuzzer. The fuzzer has a restful application program interface (API) to process inputs and outputs. When the fuzzing procedure begins, the API transfers the inputs to the controller. First, the controller inserts the event statement into the target contract source code, then deploys it on the Ethereum blockchain. The test nodes we used were Geth [14] or Ganache [15]. To shorten the block validation time, we mostly used Ganache. In the second step, according to the content and address of the test contract, the fuzzer generates the corresponding fuzzer contract and deploys it. Then, we generate the corresponding inputs according to the contract application binary interface (ABI) [16] and choose the fuzzing process according to the inputs fuzz round and the ether and function call sequences. Finally, when the input is sent, we wait for the block validation to be completed. After the validation is completed, we read the block information to understand the event occurrence and transaction content as the basis for the abnormal analysis and detection. Below, we explain in-depth the process for each step.

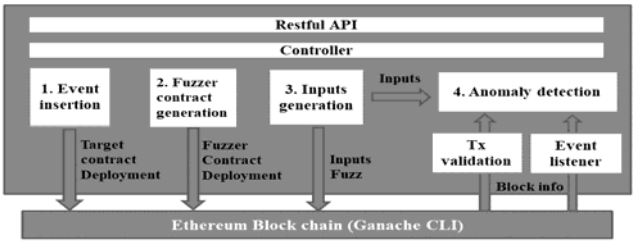

Fig. 6. Architecture of the dynamic fuzzer.

```
event event7();
event event8();
event event_return9(uint256 val);

function test(uint256 num) returns (uint256 _val)
{
    event7();
    uint256 xx = num +5 ;
    event8();
    event_return9(xx);
    return xx;
}
```

Fig. 7. Example of event insertion.

1) Event insertion: In this step, the controller parses the target contract and inserts two empty events before and after each statement to calculate the code coverage of fuzzing in each function. At the end of the function, if there is a return

statement, the controller will insert the event with the return value before the return for anomaly detection. Fig. 7 shows an example of event insertion. Here, event7 and event8 are the empty events that were inserted for code coverage, and event_return9 is the event with the the return statement parameter. The return parameter was the anomaly detection input at step four. After the event insertion finishes, the controller deploys it on the Ethereum blockchain.

2) Fuzzer contract generation: In this step, the fuzzer contract is generated according to the target contract, as shown in Fig. 8. The controller parses the target contract ABI to generate an interface for the fuzzer contract and a constructor with a parameter of the address of the previously deployed target contract. The fuzzer also uses the interface to call each function and set a fallback function with an event statement to tell the fuzzer that the fallback function was called. We considered the issue of deciding what statement should be put into the fallback function per the following two points:

a) Critical function selection: The most relevant vulnerability of the fallback function is reentrancy. The main purpose of reentrancy is to use the ether transaction statement in the fallback function to make a recursive call. There were three ether transaction statements on smart contracts in the Ethereum language Solidity: *address.transfer()*, *address.send()*, and *address.call.value().gas()()*. Thus, we select the function with these three statements and insert them into the fallback function.

b) Fallback stop condition: If a reentrancy vulnerability occurs, a recursive call to the fallback function is made. The fuzzer contract obtains all the ether of the target contract or consumes all the gas until it fails to revert. Thus, we must set a condition to stop this situation. We check that the target contract has enough ether and that the transaction has enough gas at each function call before the fallback function.

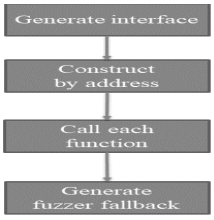

Fig. 8. Steps of fuzzer contract generation.

3) Input generation: In this step, random variables are generated according to the contract's ABI input parameter type [17], but some of the extreme values are also set in the fuzz process. In addition to the variation of input variables, the transaction gas price and limit are also part of the variation data, and the account that initiated the transaction is also included because the usage rights of the contract will often change according to the account permissions. Therefore, we set up three kinds of role account addresses: contract creater address, fuzzer contract address, and other account address. When the fuzzer initiates a transaction, these three roles are used randomly.

4) Anomaly detection: We assume that the distance between the input and output of normal and abnormal smart contract transactions will be very different. And the number of normal transactions must be more than abnormal transactions. Therefoe we design a anomaly detection method on block transcation. During the execution of the contract, the transaction event is continuously emitted. If the function has a return statement, the fuzzer also obtains the return value in the event content. The fuzzer records the input data of each transaction, which is used to calculate the Minkowski distance as in (1) with the return value. Where $X = (x_1, x_2, \dots, x_n) \in R^n$ as input data array and $Y = (y_1, y_2, \dots, y_n)\ as\ return\ value\ array \in R^n$. If return length of value array is not as same as input data array, t will fullfill with zero.

$$D(X, Y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/2} \quad \square\square\square$$

Therefore, there is a Minkowski distance for each transaction with a return value. The fuzzer collects the Minkowski distance of the same function as the parameter of the outlier analysis. In the outlier analysis approach, we use median-absolute-deviation. As a rule of thumb, a Z-score higher than 3.5 was considered a potential outlier.

TABLE III. EXPERIMENTAL RESULTS OF VULNERABILITY ANALYZER ON TEST SET

Feature Extraction Methods	***Models***	***Accuracy***	***Precision***	***Recall***	***F1-Score***
n-gram + tf-idf	**Logistic Regression**	**97.3%**	**92.9%**	**88.2%**	**90.4%**
	SVM Linear	94.8%	88.8%	80.3%	84.1%
	SVM Kernel	86.7%	63.9%	54.9%	57.2%
	K-Nearest Neighbor	93.5%	85.9%	70.2%	76.3%
	Decision Tree	95.7%	87.0%	82.0%	84.4%
	Random Forest	95.8%	95.4%	67.3%	75.1%
	Gradient Boosting	96.6%	91.0%	85.3%	87.7%
word2vec	CNN	91.4%	71.0%	58.4%	62.9%

IV. EXPERIMENT

In this section, we present our experimental results in two parts, first regarding the vulnerability analyzer and second regarding the dynamic fuzzer.

A. Vulnerability analyzer

We evaluated the effectiveness of different features and models. The training set and the test set were split from the dataset at a ratio of 80% to 20%; there were 14,383 training samples and 3,596 test samples. Because the datasets may not be balanced between positives and negatives, they were selected by stratified random sampling. Eight models and two different features were evaluated as shown in TABLE III. For the n-gram with tf-idf feature, *n* for n-gram could be one to five, and the *n* with the best score was used for evaluation.

1) Model Evalution: TABLE III. shows the average score over all vulnerabilities for each predictive model. Each vulnerability was evaluated not only by accuracy but also by precision, recall, and f1-score. The data show that the linear models, especially Logistic Regression, have better scores than complicated or ensemble models. The model trained by Logistic Regression achieved 97.3% accuracy and even an f1-score amounting to 90.4%. The CNN model with word2vec-based features did not perform well in our experiments.

```
Underflow        False
Overflow         False
Multisig         False
CallDepth        False
TOD              False
TimeDep          False
Reentrancy       True
   100%, PUSH AND PUSH DUP CALLER
     Line 17, 14: 'msg.sender.call'
```

Fig. 9. Examining result.

```
pragma solidity ^0.4.21;

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

    function HoneyPot() payable {
        put();
    }


    function put() payable {
        balances[msg.sender] = msg.value;
    }

    function get() {

        if (!msg.sender.call.value(balances[msg.sender])()) {
            throw;
        }
        balances[msg.sender] = 0;
    }


    function test(uint8 num) returns (uint8 _overflow) {
           uint8 xx = num +5 ;
           return xx;
    }

    function() {
        throw;
    }
}
```

Fig. 10. Source code of the experimental vulnerability contract.

2) Examining Results: The examining results contain three parts: whether the specific vulnerability exists, what contributes to the n-gram opcode sequences if the sample is predicted as positive, and where the vulnerabilities are located in the souce code. An example of an examining result is displayed in Fig. 9.

```
pragma solidity ^0.4.21;

contract GuessTheSecretNumberChallenge {

    bytes32 answerHash =
    0xdb81b4d58595fbbbb592d3661a34cdca14d7ab379441400cbfa1b78bc447c365;

    function GuessTheSecretNumberChallenge() public payable {
        require(msg.value == 1 ether);
    }

    function isComplete() public view returns (bool) {
        return address(this).balance == 0;
    }

    function guess(uint8 n) public payable {
        require(msg.value == 1 ether);

        if (keccak256(n) == answerHash) {
            msg.sender.call.value(address(this).balance)();
        }
    }
}
```

Fig. 11. Source code of the CTF contract.

B. Dynamic Fuzzer

Because dynamic fuzzer experiments require the parameters to be set manually according to the contract, it is difficult to conduct experiments automatically and in large quantities. To prove the fuzzer's ability, we performed three experiments with different contracts: an experimental test contract, a CTF (Capture-The-Flag) contract, and a real-world contract.

1) Experimental test contract: This test sample was a contract with reentrancy and overflow vulnerabilities and is a modified form of the honeyPotReentranceAttack [18], as

```
pragma solidity ^0.4.19;
contract Ownable
{
    address newOwner;
    address owner = msg.sender;

    function changeOwner(address addr) public onlyOwner
    {
        newOwner = addr;
    }

    function confirmOwner() public
    {
        if(msg.sender==newOwner)
        {
            owner=newOwner;
        }
    }
}

contract TokenBank is Token
{
    uint public MinDeposit;
    mapping (address => uint) public Holders;

    function Deposit() payable
    {
        if(msg.value>MinDeposit)
        {
            Holders[msg.sender]+=msg.value;
        }
    }

    function WithdrawToHolder(address _addr, uint _wei)
    public onlyOwner payable
    {
        if(Holders[_addr]>0)
        {
            if(_addr.call.value(_wei)())
            {
                Holders[_addr]-=_wei;
            }
        }
    }
}
```

Fig. 12. Source code of the TokenBank contract.

shown in Fig. 10 . The reentrancy vulnerability is in the get function and the overflow vulnerability is in the test function. In this experiment, we set the function call sequence to put, put, get, test, and the parameter of the function was generated randomly according to the ABI. We can prove that, in this experiment, the trigger of the vulnerability can be based on the corresponding parameters and account variation, as shown in TABLE IV. The reentrancy vulnerability was triggered by the get function with a fuzzer address, but it must call the put function twice before with owner and fuzzer accounts and provide ether in the transaction. When the fuzzer contract calls the get function, the fallback reentrancy detection event is emitted twice. Therefore, the reentrancy vulnerability can be detected. The overflow vulnerability was triggered by the extreme value 2**256-1, which is the acceptable maximum value of type UINT<256>. Because the fuzzer will fuzz many rounds, each round will obtain a return value from the test function. Thus, the distance between 2**256-1 and the return value will be greater than in other normal rounds due to the overflow vulnerability. Therefore, the anomaly detection can find overflow abnormalities based on such conditions.

TABLE IV. TEST CASES OF TEST CONTRACT

Call Seq.	Function	Input data		
		Parameter	*Address from*	*Ether*
1	put	(N/A)	Owner address	1 Ether
2	put	(N/A)	Fuzzer address	1 Ether
3	get	(N/A)	Fuzzer address	(N/A)
4	test	2**256 -1	Fuzzer address	(N/A)

[a.] Gas and gas price had little effect on the experiment, so they are not listed here.

2) CTF contract: The CTF contract, as shown in Fig. 11, was modified from capturetheether [19]. This sample was used to prove how SoliAudit finds the answer to Capture-the-Flag. As TABLE V. shows, the function call sequence is very simple. After initializing the contract with 1 ether, the functions are simply called in the order of guess and iscomplete. When fuzzing the correct answer 170 to guess, the function will withdraw all the ether to the caller, which makes the incomplete function return true. This causes it to look very different from the results of other rounds, thus achieving anomaly detection.

TABLE V. TEST CASES OF CTF CONTRACT

Call Seq.	Function	Input data		
		Parameter	*Address from*	*Ether*
1	Guess	(N/A)	Owner address	1 Ether
2	isComplete	170	Fuzzer address	(N/A)

[b.] Gas and gas price had little effect on the experiment, so they are not listed here.

3) Real world contract: The real world contract, called TokenBank, is shown in Fig. 12, and its address is 0x627Fa62CCbb1C1b04fFAECd72a53e37fC0E17839 [20]. There is a reentrancy vulnerability in the WithdrawToHolder function. In this experiment, SoliAudit calls the four functions described in TABLE VI. The results shows that if the owner address changes the owner to the fuzzer address, the fuzzer account can withdraw all the money by fallback function via the reentrancy vulnerability. Although this may not happen in the real world considering the owner rarely changes, it does prove that SoliAudit can detect a reentrancy vulnerability in a real-world contract.

TABLE VI. TEST CASES OF REAL-WORLD CONTRACT

Call Seq.	Function	Input data		
		Parameter	*Address from*	*Ether*
1	changeOwner	(N/A)	Owner address	(N/A)
2	confirmOwner	(N/A)	Fuzzer address	(N/A)
3	Deposit	(N/A)	Fuzzer address	1 Ether
4	WithdrawToHolder	2**256 -1	Fuzzer address	(N/A)

[c.] Gas and gas price had little effect on the experiment, so they are not listed here.

With the above three cases, we show the ability of dynamic fuzzer. In test case, we demonstrates the both integer overflow and reentrancy vulnerability with anomaly detection and attack fallback function. In CTF case, we show the fuzzer how to get the flag with anomaly detection method. In real world Token Bank case, we present fuzzer can detect reentrancy vulnerability with fallback function in WithdrawToHolder function.

V. DISCUSSION

In this section, we discuss the research results from this study.

Not all contributing opcode sequences are reasonable. In our experiments, contributing opcode sequences were derived from models trained by Logistic Regression, SVM Linear, Decision Tree, Random Forest, and Gradient Boosting, but not all were reasonable. For example, the "ORIGIN CALLER REVERT" opcode sequence may be suitable for the TxOrigin vulnerability because it contains an ORIGIN opcode. However, "ADDRESS BALANCE SUB CALL REVERT" does not seem reasonable for the TimeDep vulnerability because it does not contain the TIMESTAMP opcode, as per our expectation. This indicates that a model for a specific vulnerability may be not trained well if the contributing opcode sequence for the vulnerability is not reasonable, which could be caused by insufficient positive training samples. In our experiments, the vulnerable samples for TimeDep represented less than 7% of the dataset.

Unknown vulnerabilities. Unknown vulnerabilities are not handled by our vulnerability analyzer because the dataset must be labeled first before training. Despite this limitation, once the new vulnerabilities are identified and there are enough smart contract samples, the analyzer can adapt quickly to detect new vulnerabilities. For example, the short address vulnerability [22] is not included in our experiments because it is in the proof-of-concept stage and no real-world samples are available.

False negatives. Our method failed to detect potential vulnerabilities in some samples. For example, the inherent reentrancy vulnerability was not detected in the TokenBank smart contract located at the address 0x627Fa62C [20]. This may be caused by the removal of some relevant opcodes as stop words. If we do not remove these stop words in the data preprocessing, some otherwise missed detection samples could be correctly predicted, but the total accuracy and other scores may decrease. This suggests that the choice of which opcodes should be regarded as stop words and removed is critical. Currently, this choice is made based on our experiences, but it would perhaps be better to employ a machine learning hyperparameter to automatically choose the stop words.

True negatives. The vulnerability analyzer also helps re-label samples. Some samples are false positives in Oyente or Remix but are true negatives in the vulnerability analyzer. For example, a smart contract at the address 0x1cE7AE55 [24] was detected as having the reentrancy vulnerability. Although a malicious user can re-enter the so-called vulnerable function in the contract, the user cannot acquire extra money that does not belong to him. This behavior is not what defines the reentrancy as a vulnerability, so should be regarded as false positive. Generally speaking, audit tools prefer higher recall rates over precision rates in order to avoid missing detections of vulnerable samples. Thus, our vulnerability analyzer can provide complementary results.

Function call sequence. The function call sequence is a very important part of triggering a vulnerability by fuzzer. Just like the DAO, it may be necessary to put tokens into two different accounts and withdraw via a contract fallback function. Currently, we have no way to automatically determine the preferred call sequence. Therefore, we can only provide random or user-defined call sequences. Of course, the user-defined call sequence is a better method, but in terms of finding unknown vulnerabilities, random call sequences may yield surprising results.

Limitation. SoliAudit is not limited by the version of Solidity, as SoliAudit is compatible with any version. However, the source code must be provided if the user wants to check the vulnerability location in the source code. Otherwise, SoliAudit only requires the bytecode and the ABI to deploy on the blockchain.

VI. Conclusions

We proposed a machine learning classification based on the smart contract opcodes feature, for which we tested two kinds of feature extraction methods and eight machine learning models. The results shows that the vulnerability identification accuracy of our system is up to 90%, and our approach can rapidly adapt to new unknown weaknesses without requiring expert knowledge or pre-defined features. Moreover, SoliAudit automatically generates a fuzzer contract for fuzz testing with abnormal analysis, which successfully analyzed real-world and CTF contract cases. The dynamic fuzzer targets reentrancy and arithmetic vulnerabilities, and we believe that the dynamic fuzzer has the potential to detect unknown vulnerabilities. Moreover, it is not as difficult as it is for general fuzzer to interpret test results using our system. The dynamic fuzzer of SoliAudit can directly show the anomaly result of a function via the outlier analysis.

In the future, we plan to test dynamic opcode sequences with the vulnerability analyzer and improve the input generation mechanism using symbolic execution on the dynamic fuzzer.

References

[1] Daian, Phil. Analysis of the DAO exploit. 2016, http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. Last access, 2018.

[2] Palladino, Santiago. The Parity Wallet Hack Explained. 2017, https://blog.zeppelin.solutions/on-the-parity-wallet-multisighack-405a8c12e8f7/. Last access, 2018.

[3] Thompson, Luke. EOS Tokens Stolen In dApp Smart Contract Hack. 2018, https://ethereumworldnews.com/almost-240000-worth-of-eos-tokens-stolen-in-dapp-smart-contract-hack/. Last access, 2018.

[4] Taylor Crypto Trading ICO Hacked Out of $1.5 Million Worth of Eth, 2018. https://www.trustnodes.com/2018/05/23/taylor-crypto-trading-ico-hacked-1-5-million-worth-eth/. Last access, 2018.

[5] Proof of Weak Hands (PoWH) Coin hacked, 866 eth stolen. 2018, https://steemit.com/cryptocurrency/@bitburner/proof-of-weak-hands-powh-coin-hacked-866-eth-stolen/. Last access, 2018.

[6] Gomez, Miguel. $7.7 Million in KickCoin Stolen in Wallet Hack, Smart Contract to Blame. 2018, https://cryptovest.com/news/77-million-in-kickcoin-stolen-in-wallet-hack-smart-contract-to-blame/. Last access, 2018..

[7] NCC Group. https://www.nccgroup.trust/us/. Last access, 2018.

[8] DASP TOP 10. https://dasp.co/. Last access, 2018.

[9] Tsankov, Petar et al. "Securify: Practical Security Analysis of Smart Contracts." arXiv preprint arXiv:1806.01143 (2018).

[10] Luu, Loi et al. "Making smart contracts smarter." Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016.

[11] T.O. Bits, "Manticore - Dynamic binary analysis tool with EVM support", https://github.com/trailofbits/manticore. Last access, 2018.

[12] Nikolic, Ivica et al. "Finding the greedy, prodigal, and suicidal contracts at scale." arXiv preprint arXiv:1802.06038 (2018).

[13] Jiang, Bo, Ye Liu, and W. K. Chan. "ContractFuzzer: fuzzing smart contracts for vulnerability detection." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018.

[14] Geth. https://github.com/ethereum/go-ethereum/wiki/geth/. Last access, 2018.

[15] Truffle Suite – Ganache. https://truffleframework.com/ganache/. Last access, 2018.

[16] ABI of Ethereum Smart Contracts. https://github.com/ethereum/wiki/wiki/Ethereum-ContractABI/. Last access, 2018.

[17] ABI specification of Ethereum smart contracts. https://solidity.readthedocs.io/en/latest/abi-spec.html/. Last access, 2018.

[18] HoneyPotReentranceAttack. https://github.com/gustavoguimaraes/honeyPotReentranceAttack/. Last access, 2018.

[19] Capturetheether. https://capturetheether.com/challenges/. Last access, 2018.

[20] TokenBank Smart Contract. https://etherscan.io/address/0x627Fa62CCbb1C1b04fFAECd72a53e37fC0E17839/. Last access, 2018.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Efficient Estimation of Word Representations in Vector Space," arXiv:1301.3781 (2013).

[22] Vessenes, Peter. The ERC20 Short Address Attack Explained. 2017, https://vessenes.com/the-erc20-short-address-attack-explained/. Last access, 2018.

[23] SoliAudit Vulnerability Analyzer Dataset. https://goo.gl/UAUpK5/. Last access, 2018.

[24] 0x1cE7AE55 Smart Contract. https://etherscan.io/address/0x1cE7AE555139c5EF5A57CC8d814a867ee6Ee33D8/. Last access, 2018.

[25] OWASP TOP 10 Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project/. Last access, 2018.

[26] Remix, Ethereum-IDE. https://remix.ethereum.org/. Last access, 2018.

[27] Term frequency–inverse document frequency. https://en.wikipedia.org/wiki/Tf%E2%80%93idf. Last access, 2018.

[28] Source Code of SoliAudit. https://github.com/jianwei76/SoliAudit. Last access, 2019