SCScan: A SVM-based Scanning System for Vulnerabilities in Blockchain Smart Contracts

Xiaohan Hao
School of Computer Science
China University of Geosciences
Wuhan, P.R. China

Wei Ren*
School of Computer Science
China University of Geosciences
Wuhan, P.R. China
Guangxi Key Laboratory of Cryptography and Information Security
Guilin, P.R. China 541004
Key Laboratory of Network Assessment Technology, CAS
(Institute of Information Engineering, Chinese Academy of Sciences,
Beijing, P.R. China 100093)
Email: weirencs@cug.edu.cn

Wenwen Zheng
School of Computer Science
China University of Geosciences
Wuhan, P.R. China

Tianqing Zhu
School of Computer Science
China University of Geosciences
Wuhan, P.R. China

***Abstract*—The application of blockchain has moved beyond cryptocurrencies, to applications such as credentialing and smart contracts. The smart contract allows ones to achieve fair exchange for values without relying on a centralized entity. However, as the smart contract can be automatically executed with token transfers, an attacker can seek to exploit vulnerabilities in smart contracts for illicit profits. Thus, this paper proposes a support vector machine (SVM)-based scanning system for vulnerabilities on smart contracts. Our evaluation on Ethereum demonstrate that we achieve a identification rate of over 90% based on several popular attacks.**

***Index Terms*—Blockchain; Ethereum; Smart Contract; Vulnerability Detection; Support Vector Machine**

I. Introduction

Blockchain is increasingly popular, and has many applications in a broad range of industry sectors [1]–[3]. One popular instantiation of blockchain is smart contract, which has also been used in different settings [4]–[6]. A smart contract can support automated programming execution in a distributed fashion. Similar to many other technologies, including blockchain-based technologies, there are a number of limitations and challenges associated with smart contracts. For example, there have also been a number of known attacks exploiting vulnerabilities in smart contracts, which will also be addressed by this paper. For example, in June 2016, the DAO (Decentralized Autonomous Organization) vulnerability reportedly resulted in loss of 50 million USD [1]. Hence, smart contract security has been the subject of recent focus [5], [7]–[14]. Of the existing smart contract security analysis approaches, statically analyzing smart contract code remains a somewhat efficient way to identify vulnerabilities from source codes on a large scale.

In this paper, we propose a security scanning system for smart contracts. The system is focusing on several popular attacks such as reentry attacks, access control vulnerabilities, integer overflow vulnerabilities, unsafe function return value verification, denial of service attacks, predictable random number attacks, and unknown possible vulnerabilities. Based on the scanning outcome, an in-depth security rating is reported based on the threat degree of each vulnerability. We use the metrics described in the "Blockchain Smart Contract Audit Security Report" issued jointly by Blockchain Security Research Center and China Blockchain Application Research Center[2]. In addition, our proposed system also provides batch scanning function for large scale smart contracts. Specifically, in our approach we leverage support vector machine (SVM) to improve the detection of unknown vulnerabilities.

The rest of the paper is organized as follows. Section II reviews relevant prior work. In Section III, we introduce the building blocks in our proposed approach. Section IV explains one can perform data cleaning and identify vulnerabilities in smart contracts using our approach. In Section V, we evaluate the performance of our proposed approach. Finally, Section VI concludes this paper.

II. Related Literature

There have been increased focus on blockchain research, ranging from designing new cryptographic primitives to sup-

1. https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html, last accessed Oct 29, 2019.

2. http://www.caict.ac.cn/kxyj/qwfb/bps/201901/P020190111354077196849.pdf, last accessed Oct 29, 2019.

978-1-6654-0392-4/20/$31.00 ©2020 IEEE
DOI 10.1109/TrustCom50675.2020.00221

port different functions to identifying vulnerabilities in smart contracts, and so on. Atzei et al. [4] analyzed the security vulnerabilities of Ethereum, and presented a taxonomy of common programming pitfalls that may lead to these vulnerabilities. Bartoletti et al. [5] presented a comprehensive survey of Ponzi schemes on Ethereum, and analyzed their behaviors and impacts. Chen et al. [15] studied Solidity, a recommended compiler and language for smart contracts, and revealed that it fails to optimize gas-costly programming patterns. They then proposed and developed GASPER, a new tool for automatically locating gas-costly patterns by analyzing bytecodes in smart contracts. Luu et al. [10] proposed to enhance the operational semantics of Ethereum and reduce the vulnerability of contracts. Bragagnolo et al. [7] addressed the lack of ability of checking contracts by analyzing status using decompression techniques based on contract structure definitions. Their solution uses a mirror-based architecture to represent object responsible for interpreting the state of contracts. In addition, Knecht et al. [16] proposed a smart contract deployment and management platform that can reliably execute development and code quality tools.

Ethereum smart contracts are written in Solidity, and shipped to blockchain in the Ethereum Virtual Machine (EVM) bytecode format. Grishchenko et al. [17] proposed the first complete small-step semantics of EVM bytecode and formalized the proof assistant. This allows them to obtain the executable code, which can then be used to validated against official Ethereum test suite. Tikhomirov et al. [18] provided a comprehensive taxonomy of code problems and implemented an extensible static analysis tool, SmartCheck, which can convert Solidity source codes into an XML-based intermediate representation and check the representation against XPath patterns. Hegeds [19] proposed using the "OO" metrics for Solidity smart contracts, and analyzed more than 40,000 Solidity source files with the prototype tool.

Due to the renewed interest in artificial intelligence (AI), broadly defined to include both machine learning and deep learning, there are also rising some attempts to utilize AI techniques in smart contract vulnerability detection, as evidenced by the approaches of Jiang et al. [9], Mavridou and Laszka [11], Wu et al. [20], Ye et al. [21], and so on. However, some of their methods can not detect unknown vulnerabilities or evaluate and classify contracts after detecting vulnerabilities, which is not convenient for large scale contracts. In this paper, we proposes a support vector machine (SVM)-based scanning system for vulnerabilities on smart contracts, including using SVM to identify DoS attacks in smart contract and learning from SVM and forecasting new vectors.

III. Building Blocks

In this section, we will introduce the various building blocks underpinning our proposed system.

A. System Model

Our proposed system is designed for Ethereum, as well as for smart contracts. Hence, our system is designed to be as flexible as possible, and a high-level overview is shown in Fig. 1.

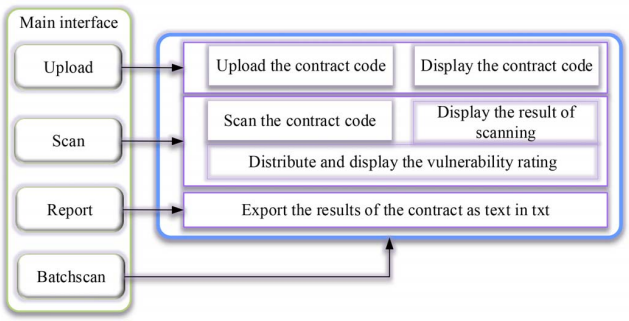

Fig. 1. High-level overview of SVM-based Scanning System

The user interface facilitates users to check contract codes with corresponding vulnerabilities, and to locate their problematic codes directly. The interface provides four functions in two modules. In the first module, one can upload files, perform scanning, and export reports. In other words, this module allows the user to locate the vulnerability in smart contract (down to a specific code / line), present the corresponding vulnerability name, and display the vulnerability rating. The second module is also known as the batch scanning module, which allows a number of smart contract codes in a designated directory to be scanned. After the system has scanned each contract, it will export the result.

1) Single Scanning Module: After users upload the contract of interest to the system via the file upload function, the system will clean the contract code first, before scanning the cleaned code for the following: re-entry vulnerability, access control vulnerabilities, integer overflow vulnerabilities, unsafe return validation vulnerabilities, denial of service attack vulnerabilities, and predictable random number vulnerabilities. Next, if applicable, the module will return vulnerability location to users. In addition, the system classifies the security of scanned contract and outputs results in different colors according to vulnerabilities. The specific structure is shown in the left side of Fig. 2, and on the right side is the batch contract scanning module.

2) Batch Scanning Module: In order to save the time and cost of detecting a large number of contract codes, we design batch scanning module, which users are required to place the scanned codes into folders specified by the system. The system will first clean the data of each contract code, and then scan the code based on the six attack folders.

3) Six Attack Methods: In this module, the vulnerability scanning includes six types of attacks, and the detailed description of above six attacks are shown in Section III-B.

4) Level Evaluation Criteria: In order to make the user more intuitive to observe the results, we use level evaluation criteria module. While identifying each vulnerability, the contract is scored for risks according to an internal scoring mechanism. E.g., 0-5 is a low-risk vulnerability, as shown

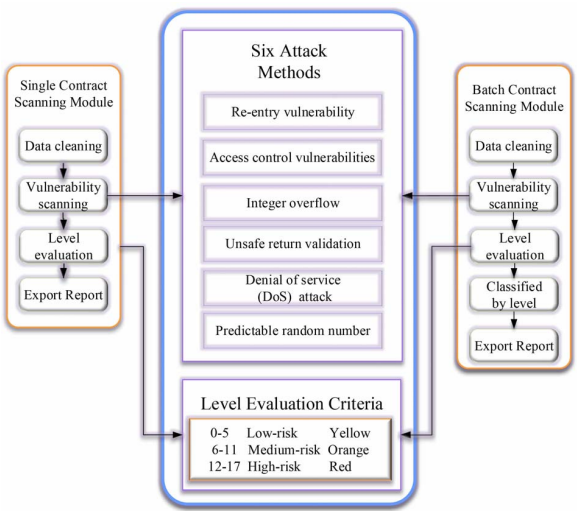

Fig. 2. Smart Contract Scanning Modules

in yellow; 6-11 is a medium-risk vulnerability, as shown in orange; 12-17 is a high-risk vulnerability, as shown in red.

5) Support Vector Machine (SVM): SVM is a supervised learning method, where a decision boundary is determined after the model has learned and calculated the position of each point in the black and white lists. After determining the decision boundary and the decision vector, we introduce the test variable and let it multiply with the decision vector to see if the result will exceed the decision boundary for prediction. In this paper, we use SVM to detect contract code based on known vulnerability reports, so that users can clearly and intuitively see their contract code and corresponding vulnerabilities, and help users locate their code problems faster so as to facilitate rapid modification.

B. Adversary Model

In this section, we identify six potential vulnerabilities that can be exploited by attackers as follows:

Re-entry attacks: In our scenario, re-entry attacks may be triggered when the contract sends an Ether to an unknown address. The attacker can build an attack contract on its own external address, and the Fallback function in the smart contract that exists at that address can construct malicious codes. When the contract sends a currency request to this address, it calls the Fallback function to start the malicious code forged by the attacker. The developer will call the coin function when designing the contract. When using the "call.value()()" function to transfer the currency, the function will give the remaining gas all to the externally called Fallback function (transfer and send). The coin function will only provide 2300 gas for external calls, and the attacker can deploy an action similar to the presence of a malicious recursive call at the location of the target address to bring up the Ethereum in the contract wallet, causing huge losses.

Access control vulnerabilities: There are two ways to call the underlying "call()" and "delegatecall()" in the process of writing smart contracts. For example, contract "A" calls "fun" function of contract "B". When "A" uses "call" function, it will jump to the external contract "B" to execute the corresponding function in "B" contract. When the function is executed, it will jump back to contract "A" to execute the further code.

Integer overflow vulnerabilities: There is also an integer overflow problem in writing smart contracts. The data types we generally use in contracts are unsigned integer ($uint$) and signed integer ($init$). When the variable declared by $uint$ reaches the maximum value of 255, adding 1 will result in being 0 and consequently leads to an error in the entire calculation. The range of the integer type is [-128,127], where the highest bit represents the symbol bit and is not counted. When the variable reaches the maximum value of 127, adding 1 will flip to -128, and the preceding sign bit will be changed from 0 to 1. That is, the corresponding expressions are: 255 + 1 = 0 and 127 + 1 = -128. Developers often use the require function in the development process to determine whether the balance in the current wallet has sufficient balance (balances[msg.sender] - amount $>$ 0). Through the above analysis, we know that even if the amount to be deducted is greater than the user's balance, the verification will be successful. The result is not only the extracted Ethereum is greater than the user's balance, but also the balance is updated to an erroneous number. In addition, in the preparation of the contract, var is an 8-bit unsigned integer, and if the threshold value of the variable i is limited to a 256-bit variable, it may also cause i to cross the boundary in the continuous loop superimposition and lead to the program failing to operate normally.

Unsafe function return value verification: There are many functions in smart contracts that have problems with return value validation. On the one hand, the underlying call function delegatecall function returns a *bool* type value after the underlying call terminates and displays the outcome of the call result (i.e., success or failure). The callcode function and the call function can also return a *bool* type value after the call terminates to display the result. We remark that the transfer function call.value()() and send() functions also return true or false at the end of the call to indicate the result of the call. Thus, an attacker can call function send() for the currency operation, but does not determine the return value of the function. That is to say, if the function fails, the Ethereum is not sent to our desired address, but the user's account balance is less. To prevent the trigger of this vulnerability, the function transfer() can be used to perform the currency operation, while this function does not return a value but automatically rolls back the state when the call fails.

Denial of service (DoS) attacks: A classic open auction scenario is as follows: if a new bidder appears with the current highest bid price, (s)he will replace the original highest bidder and become the current winning bidder for the auction. The contract will automatically return the previous bidder's Ethercoin to the price address through using send() function. Through the above analysis, we know that when the contract calls the send() function, the Fallback function will

be triggered. The attacker can build attack codes in order to achieve this purpose. Specifically, when the transfer request is initiated (returning the previous successful bidder's Ethercoin), the Fallback function of the external contract is executed, and the revert() function is written in the Fallback function so that the malicious operation can be performed to cause contract to run error actively. Consequently, it will result in a transfer failure and other users cannot become the winning bidder. Hence, the attacker can win the bid at a lower price.

Predictable random number attacks: Private seed variables may be used in smart contracts and all data in the blockchain is public. Thus, the attacker can get the value corresponding to the seed variable, and the random variable becomes transparent. Hence, there is a risk that the variable can be exploited. Besides, some developers also use $block.blockhash(unitblocknumber)$ to get a random hash value, which will result in a final result of an invalid hash value $0x0000....$

IV. Proposed System

There are three modules in our proposed system: single smart contract scan, batch smart contract scan, and using SVM to identify DoS attacks in contracts. Besides, we sum up a list of six vulnerabilities in smart contracts and how the SCSCAN will behave as shown in Table I.

A. Single Smart Contract Scanning

1) Data Cleaning: In our experiments, it is necessary to clean the data of different types of contracts and converted them into a uniform code format. The pseudocode is presented in Algorithm 1.

Algorithm 1: Data Cleaning

```
for i in range(code) do
    /*Detect // comment and filter*/
    if check_obliquel1(i) then
        delete_obliquel1(i);
    /*Detect /**/ comment and filter*/
    if check_obliquel2(i) then
        delete_obliquel2(i);
    /*Detect blank lines and filter*/
    if check_line(i) then
        delete_line(i);
    /*Add the filtered code data to the line label*/
    add_num(i);
```

The operations required are as follows:

1) Identify comments with $//(doubleslash)$ and remove them.
2) Identify and remove all comments with $/**/(slashstars)$.
3) Identify blank lines and delete them.
4) Identify the blanks before and after each line and delete them. In oder to ensure the unrepeatability of key value, we need to add the number of lines corresponding to each line to the back of codes to form a new key value.

2) Recognition of Re-entry Vulnerabilities: According to the previous introduction, it is worth to know that a total of three coin transfer functions in the smart contract are $call.value()$, $send()$, and $transfer()$. The first function in the transfer process will open all the gas provided in the contract to the Fallback function of the external address, which is easy to be hijacked by attackers. Therefore, we will perform pattern matching on the call.value function. Once found there is a called in the contract for transfer, the location is identified as a re-entry attack and its vulnerability information and location are output.

3) Access Control Vulnerability Identification: According to the previous introduction, the access control vulnerability is due to the use of the delegatecall function when the underlying call is made, and the function uses $msg.data$ to transfer the parameters. $msg.data$ (external caller passing information) is a mutable message that can be written according to external caller's wishes. Therefore, some malicious operations may be written, which violates the developer's intended operation. In our system, we need to match the variable information of delegatecall function and $msg.data$ in the process of identification. If we find these two threats in the code, we can locate them as access control vulnerabilities and output their vulnerability information and location.

4) Integer Overflow Vulnerability Identification: There are two cases of integer overflow vulnerability, on the one hand, if the data types are same, the forced addition and subtraction operation are performed because there is no operability of the judgment operation, which is similar to 255 + 1 = 0. On the other hand, these two variables are declared by different data types, one is an 8-bit integer and the other is a 256-bit integer, and such an 8-bit may have a situation in which it is never possible to exceed the 256-bit integer during the accumulation comparison and cause an out-of-bounds, which may cause the program to execute incorrectly. According to the above analysis, we make the following two identification methods, the pseudo code is as follows in algorithm 2.

5) Insecure Function Return Value Validation Vulnerability Identification: In order to identify the vulnerability, we need not only to find the function that may trigger the vulnerability through pattern matching. After finding the corresponding functions, we make the following two checks, the specific algorithm of identifying is shown in algorithm 3.

6) Predictable Random Number Vulnerability Identification: According to the previous introduction, this vulnerability involves two ways, one is a private seed variable, and the other is a random hash based on blockhash. In order to detect these two kinds of vulnerabilities, the following two tests are performed:

1) Directly match the private seed (seed) to check whether it exists, if exists, then report the vulnerability directly.
2) Identify whether developer uses the blockhash function to get the hash value. If used, checking whether the block number used in the function is the current block

Algorithm 2: Integer Overflow Vulnerability Identification

```
for i in range(key) do
    /*Check if there is a special character <> −+ =/
    if i.find(> or < or - or + or =) then
        /*Check if the line has 0*/
        i.check(0);
    /*Check if there are two keywords 'for(' and 'for ('*/
    if i.find(for( or for ( and var) then
        /*Extract the parameter and assign it to m*/
        m = re.findall(''.* < (.*); .*'', i);
    /*Check if the parameter range uses the length
    function*/
    if m.find(.length) then
        record its position;
    else
        /*Check if the type of m is unit*/
        if check(m,unit) then
            record its position;
    else
        /*Iterative search in the code if you can't find it*/
        find(m);
```

Algorithm 3: Insecure Function Return Value Validation Vulnerability Identification

```
for i in range(key) do
    /*Find five key functions*/
    if find(delegatecall call callcode send call.value) then
        /*Check if the function is assigned and whether it
        is bool*/
        check_bool(i);
    else
        /*Check whether key functions are nested in if or
        require functions*/
        if check_judge(i) then
            /*This contract is secure*/
            pass;
```

number. After identifying the above problems, it is shown that there may exist predictable random number vulnerabilities, which can be located and output.

7) Rating and Output of Contract Code Vulnerabilities: After completing each vulnerability identification, a list will be returned. Next, calculating the score of contracts, then classifying them according to the score range conforming to the security level. Different levels are displayed in different colors according to scores during the output process. That is, 0 to 5 for low-risk vulnerabilities are shown in yellow, 6-11 for medium-risk vulnerabilities are shown in orange, and 12-17 are high-risk vulnerabilities in red. Specifically, we assign different hazard index to different vulnerabilities (except for predictable random number vulnerabilities, which hazard index is set to 2, the other five vulnerabilities are set to 3), and these values belong to our system settings, which can be adjusted according different situations. The specific calculation formula is as follows:

$$Score = (RV + AV + IOV + UV + DV) * 3 + PV * 2$$

In above formula, RV, AV, IOV, UV, DV and PV represent the number of corresponding vulnerabilities respectively.

B. Batch Smart Contract Scanning

Batch smart contract scanning is similar to single smart contract scanning in general process. It also needs to experience from data cleaning to identify six vulnerabilities. After the identification is completed, the batch smart contract scanning module does not need to output all the vulnerability information, but needs to count the number of security contracts and the number of contracts with vulnerabilities, and then calculate the threat rate of the smart contracts. At the same time, it is necessary to classify vulnerabilities according to each vulnerability and integrate the same level of vulnerabilities.

At the end of identification, users can export the scanning report, which helps them locate the specific vulnerability contract according to the details of identification, and gives them the priority to check the contract vulnerability according to the security level of the contract. At the same time, a single smart contract vulnerability module is used to perform further vulnerability identification on a specific contract, helping developers to locate a specific line and a certain vulnerability.

TABLE I
SIX VULNERABILITIES IN SMART CONTRACTS AND DETECTION METHODS

Six vulnerabilities in smart contracts	Available Solutions
Re-entry Vulnerabilities	Identify the call. value() () function
Access Control Vulnerability	Identify the delegatecall function and the msg.data function
Integer Overflow Vulnerability	Indentify the special character <>+ = and check if there is 0 in this line Identify the for loop and check whether the parameter range uses the length function
Insecure Function Return Value Validation Vulnerability	Identify whether the function is assigned and bool Identify functions nested in if or require functions
Predictable Random Number Vulnerability	Match private seed Identify whether the blockhash function is used to obtain the hash value and check whether the block number used in the function is the current one

C. Using SVM to identify DoS attacks in contracts

1) Vector modeling: Through analysis of the Ethereum smart contract's denial of service attack analysis, a five-dimensional vector can be constructed for learning recognition from the following five aspects: whether the Fallback function exists, the number of lines of the Fallback function, the number of special characters <, the number of >, the number of three transfer functions(call.value, send, transfer) and whether the user can retrieve the tendered sum by himself.

2) Data Cleaning: Since vector construction is needed, we need to cleaning data again after the previous cleaning. From the previous analysis of DoS attacks, we can conclude that most of these attacks exist in Fallback functions and all have payable tags. In the second data cleaning process, we need to identify payable function. After finding this function, all the contents of the function are extracted and stored in a new list. Next, in the process of searching and identifying, we only need to detect in the new list, which saves a lot of cost and time and improves the efficiency and accuracy of identification.

3) Constructing Vector: After data cleaning, we can start to construct our own vectors, which can be divided into the following five steps, the specific algorithm of constructing vectors is shown in algorithm 4.

The specific operation method is as follows:

1) Detect whether there is a Fallback function (payable), which is recorded as 1 if exists and 0 if no exists.
2) Check the number of lines in the new list after the data has been cleaned and record it. The code with this vulnerability has a certain number of lines, while some contracts only take extremely simple operations of two or three lines in the Fallback function.
3) The total number of <, > is identified and recorded. When the vulnerability exists, these two special characters will be used to compare the current bid amount with the previous one.
4) Identifying and counting these three transfer functions: call.value, send, and transfer.
5) In the operation judgment of the last user to retrieve the bid amount, we need to judge from the following two aspects:
   (1) First, checking whether the contract uses the 'mapping' data structure in the writing process. If it exists, the variable declared by the data structure is stored in the list. If it does not exist, then the operation record of the user's own withdrawal amount can be considered as 0.
   (2) The variable that has the mapping data structure declaration matches in the Fallback function to check if there is the same variable and whether the plus sign is used at the same time. If both of the above two conditions are met, it can be considered that there is an operation record of 1 for the user to retrieve the amount, otherwise, recording as 0.

4) Learning through SVM and Predicting New Vectors: The SVM (support vector machine) algorithm is a supervised learning method that finds a decision boundary by learning and calculating the location of points in the black and white list. Due to the requirement for supervised learning, we need to find the same amount of whitelists and blacklists from the code crawled by the Etherscan official website through manual auditing. The basic idea of SVM learning is to solve the separation hyperplane which can divide the training data set correctly and has the largest geometric interval. For linearly separable data sets, there are infinite hyperplanes, but the separated hyperplanes with the largest geometric spacing are unique. The red variable on the left is negative variable, and the blue variable on the right is positive variable. What SVM needs to do is to find a decision boundary in the middle, and then find the widest distance by extending the boundary outward.

Algorithm 4: Constructing Vector

```
for i in range(code) do
    /*Find the Fallback function*/
    if find(payable function) then
        /*If exits, recording it as 1*/
        vector.append(1);
        break;
    else
        /*If not exits, recording it as 0*/
        vector.append(0);
        /*Record the number of the Fallback function's
        rows*/
        vector.append(len(code));
for i in range(code) do
    count1 = 0;
    /*Find special characters <> and count them*/
    if find(<>) then
        count1++;
        vector.append(count1);
for i in range(code) do
    count2 = 0;
    /*Find the transfer functions and count them*/
    if find(call.value,send,transfer) then
        count2++;
        vector.append(count2);
for i in range(code) do
    /*Identifying the mapping data type*/
    if check(mapping) then
        /*Check for the presence of +*/
        if find(+) then
            vector.append(1);
        else
            vector.append(0);
        else
            /*This contract is secure*/
            pass;
```

In other words, the purpose of SVM is to find the widest gap to divide two types of variables.

SVM is a kind of supervised learning, the specific process includes data cleaning, data markup, characterization, data splitting(training data and test data). After training a model through the former data, we can verify this model by testing data and judging whether it meets expectations. If the answer is no, then adjusting model until approach our expectations. The specific codes are shown in V.

V. Performance Analysis

The system is written on the Mac platform through using Python 3.7. We test our system with two different types: Macbookair 2014 and Macbookpro 2018.

Firstly, in order to understand our method more intelligible and acceptable, we elaborate part of our code when learning through SVM and predicting new vectors. The loss function can quantify the classified loss, and its mathematical meaning can be obtained in the form of 0-1 loss function. After the vector construction of the black and white list, we need to construct the learning vector and data marking, so that the white list corresponds to 0 and the black list to 1. The data marking code is as follows:

$$contract\_vec = np.r\_[w\_vec, b\_vec]$$

$$mark = [0] * len(w\_vec) + [1] * len(b\_vec)$$

The constructed vector is passed into SVM function for learning to get decision boundary and decision vector. The code of training data is as follows:

$$clf = svm.SVC(kernel =' linear')$$

$$clf.fit(contract\_vec, mark)$$

After the training, the predict function is called to pour into the contract vector tested to obtain its output. If the output is shown as 0, the predicted result is that there is no denial of service attack; if it is shown as 1, it means there may exist denial of service attacks. The code to validate models is as follows:

$$Test = clf.predict([Svm\_dataclean(key)])$$

$$metrics.accuracy\_score(Test, source)$$

Vulnerability Recognition Rate In our experiments, we test the recognition rate of six types of vulnerabilities. To evaluate the performance of our scheme, we implement a system which can not only scan smart contracts to identify vulnerabilities, but calculate corresponding recognition rates. The specific results are shown in Table II. In this table, we can conclude that all identification rates (the number of contracts with vulnerabilities identified by our system / the total number of contracts with vulnerabilities) except DoS attacks have reached 100%. The recognition rate of DoS attacks depends on SVM and the probability of SVM recognition is largely depending on the establishment of black-and-white lists. Therefore, this recognition rate(92%) is acceptable.

TABLE II
Vulnerability Recognition Rate

Vulnerability name	Recognition rate
Reentry attacks	100%
Access control vulnerabilities	100%
Integer overflow vulnerabilities	100%
Unsafe function return value verification	100%
Denial of service attacks	92%
Predictable random number attacks	100%

Applicability Besides, we compare our scheme with the other four schemes from six perspectives: whether propose identification methods of smart contracts, detect unknown vulnerabilities, return vulnerability location to users, identifying vulnerabilities based on source code, identifying vulnerabilities based on behavior, level evaluation and classify contracts. After comparison, we can conclude that our system meet all above conditions, while others schemes can only meet parts of them. Therefore, our system has good applicability and user-friendliness. The specific results are shown in the Table III.

Efficiency In the course of testing, the module scanned 1000 smart contracts randomly crawled from the Etherscan system, including 303 contracts with vulnerabilities and 697 security contracts, and the threat rate (the number of contracts identified with vulnerabilities / the total number of contracts scanned) was 30.3%. From the effect of this scanning, it is quite remarkable that the vulnerable contracts can be found and positioned accurately.

Time complexity and space complexity We mainly consider the resources and time consumed by the algorithm from the two dimensions of the time and space occupied by the algorithm. According to the code, we can conclude that the time complexity is O(n) and the space complexity is also O(n), which means the amount of computer resources required to run our code is acceptable.

VI. Conclusion

In this paper, we proposed an Ethereum Smart Contract Scanning System – SCSCAN. In the recognition process of the system, not only the pattern matching method is used, but the supervised learning method is added to solve the loopholes existing in the connection between some logic codes, so as to improve the effectiveness of scanning. We also demonstrated the potential of SCSCAN based on our evaluations.

In the future, we intend to expand our own sample space and include deep learning approaches to more effectively disassemble and analyze the contract code. We also intend to extend our proposed system to scan for vulnerabilities of other smart contracts.

Acknowledgement

The research was financially supported by National Natural Science Foundation of China (No.61972366), the Founda-

TABLE III
SCHEME COMPARISON

	Our scheme	Giancarlo Bigi [1]	Santiago Bragagnolo [7]	Contract Fuzzer [22]	Oyente [10]
Propose identification methods of smart contracts	✓	✓	✓	✓	✓
Detect unknown vulnerabilities	✓	×	×	×	×
Return vulnerability location to users	✓	×	×	✓	✓
Identifying vulnerabilities based on source code	✓	×	✓	✓	✓
Identifying vulnerabilities based on behavior	✓	✓	×	✓	✓
Level evaluation and classify contracts	✓	×	×	×	×

tion of Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences (No. KFKT2019-003), the Foundation of Guangxi Key Laboratory of Cryptography and Information Security (No. GCIS201913), and the Foundation of Guizhou Provincial Key Laboratory of Public Big Data (No. 2018BDKFJJ009, No. 2019BDKFJJ003, No. 2019BDKFJJ011).

REFERENCES

[1] G. Bigi, A. Bracciali, et al., Validation of decentralised smart contracts through game theory and formal methods, in: Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday, 2015, pp. 142–161.

[2] F. Hawlitschek, B. Notheisen, T. Teubner, The limits of trust-free systems: A literature review on blockchain technology and trust in the sharing economy, in: Electronic commerce research and applications, Vol. 29, Elsevier, 2018, pp. 50–63.

[3] P. J. Taylor, T. Dargahi, et al., A systematic literature review of blockchain cyber security, in: Digital Communications and Networks, Vol. 6, Elsevier, 2020, pp. 147–156.

[4] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), in: International conference on principles of security and trust, Springer, 2017, pp. 164–186.

[5] M. Bartoletti, L. Pompianu, An empirical analysis of smart contracts: platforms, applications, and design patterns, in: International conference on financial cryptography and data security, Springer, 2017, pp. 494–509.

[6] P. Siano, G. De Marco, et al., A survey and evaluation of the potentials of distributed ledger technology for peer-to-peer transactive energy exchanges in local energy markets, in: IEEE Systems Journal, Vol. 13, IEEE, 2019, pp. 3454–3466.

[7] S. Bragagnolo, H. Rocha, et al., Smartinspect: solidity smart contract inspector, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, 2018, pp. 9–18.

[8] J. Feist, G. Grieco, A. Groce, Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE, 2019, pp. 8–15.

[9] B. Jiang, Y. Liu, W. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 259–269.

[10] L. Luu, D. H. Chu, et al., Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254–269.

[11] A. Mavridou, A. Laszka, Designing secure ethereum smart contracts: A finite state machine based approach, in: International Conference on Financial Cryptography and Data Security, Springer, 2018, pp. 523–540.

[12] R. M. Parizi, A. Dehghantanha, et al., Empirical vulnerability analysis of automated smart contracts security testing on blockchains, in: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2018, pp. 103–113.

[13] C. F. Torres, J. Schütte, et al., Osiris: Hunting for integer bugs in ethereum smart contracts, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACM, 2018, pp. 664–676.

[14] S. Wang, C. Zhang, Z. Su, Detecting nondeterministic payment bugs in ethereum smart contracts, in: Proceedings of the ACM on Programming Languages, Vol. 3, ACM, 2019, pp. 1–29.

[15] T. Chen, X. Li, et al., Under-optimized smart contracts devour your money, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 442–446.

[16] M. Knecht, B. Stiller, Smartdemap: A smart contract deployment and management platform, in: IFIP International Conference on Autonomous Infrastructure, Management and Security, Springer, 2017, pp. 159–164.

[17] I. Grishchenko, M. Maffei, C. Schneidewind, A semantic framework for the security analysis of ethereum smart contracts, in: International Conference on Principles of Security and Trust, Springer, 2018, pp. 243–269.

[18] S. Tikhomirov, E. Voskresenskaya, et al., Smartcheck: static analysis of ethereum smart contracts, in: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2018, pp. 9–16.

[19] P. Hegeds, Towards analyzing the complexity landscape of solidity based ethereum smart contracts, in: the 1st International Workshop, 2018, pp. 35–39.

[20] F. Wu, J. Wang, et al., Vulnerability detection with deep learning, in: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), IEEE, 2017, pp. 1298–1302.

[21] C. K. Frantz, M. Nowostawski, From institutions to code: Towards automated generation of smart contracts, in: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W), IEEE, 2016, pp. 210–215.

[22] B. Jiang, Y. Liu, W. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2018, pp. 259–269.