ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts

Wei Wang, Jingjing Song, Guangquan Xu, *Member, IEEE*, Yidong Li, Hao Wang, and Chunhua Su

***Abstract*—Smart contracts are decentralized applications running on Blockchain. A very large number of smart contracts has been deployed on Ethereum. Meanwhile, security flaws of contracts have led to huge pecuniary losses and destroyed the ecological stability of contract layer on Blockchain. It is thus an emerging yet crucial issue to effectively and efficiently detect vulnerabilities in contracts. Existing detection methods like Oyente and Securify are mainly based on symbolic execution or analysis. These methods are very time-consuming, as the symbolic execution requires the exploration of all executable paths or the analysis of dependency graphs in a contract. In this work, we propose ContractWard to detect vulnerabilities in smart contracts with machine learning techniques. First, we extract bigram features from simplified operation codes of smart contracts. Second, we employ five machine learning algorithms and two sampling algorithms to build the models. ContractWard is evaluated with 49502 real-world smart contracts running on Ethereum. The experimental results demonstrate the effectiveness and efficiency of ContractWard. The predictive Micro-F1 and Macro-F1 of ContractWard are over 96% and the average detection time is 4 seconds on each smart contract when we use XGBoost for training the models and SMOTETomek for balancing the training sets.**

***Index Terms*—Blockchain, machine learning, smart contracts, vulnerability detection.**

I. Introduction

THE concept of smart contract was first proposed by Nick Szabo in 1990 s, defined as "A smart contract is a computerized transaction protocol that executes the terms of contract" [1]. Nevertheless, at that time, the exploration only stayed at theoretical level due to the lack of trusted execution environments. Since 2009, with the emergence of blockchain technology that was first applied in Bitcoin [2], a reliable execution environment has been offered to smart contracts. The remote point-to-point value delivery can be realized without any trusted third party through the integration of many technologies such as distributed data storage, consensus protocols and encryption algorithms. However, as Bitcoin system is not Turing-complete, it is unable to handle complex business logic via smart contracts.

Inspired by Bitcoin, Vitalik Buterin developed Ethereum [3] in 2013. Ethereum is an open-source distributed computing platform and operating system based on blockchain, which features smart contracts. To meet various demands of business scenarios, Ethereum provides a Turing-complete Ethereum Virtual Machine (EVM) [4] that enables developers to deploy decentralized applications (Dapps) on it. Dapps are also called smart contracts. They are widely applied in many fields like financial services [5], infrastructures [6], internet of things [7], health care [8], [9], and others [10], [11]. By the end of May 2018, the cryptocurrency market capital has reached about 35 billion US dollars on Ethereum [12]. Blockchain as a newly developed technology is vulnerable [13] in terms of lacking regulations and programmable characteristics. These vulnerabilities can be easily exploited, resulting huge losses. For instance, the DAO bug [14] resulted in losses of over 60 million US dollars in June 2016. Besides, Integer Overflow bug on the BEC campaign caused instantaneous evaporation of over $900 million.

Different from traditional applications in Andriod, smart contracts are automatically executed in short codes, and thus their running process cannot be manually interfered. In addition, contracts often need to reward miners with cryptocurrency [15]. At present there are many types of cryptocurrency on Ethereum and the economic values of these cryptocurrencies are incalculable. Due to running on distributed nodes on the blockchain, once a contract is executed, it is almost impossible to upgrade or modify the contract. The data generated by contracts on blockchain is also hard to be modified. Therefore, vulnerability detection [16] is indispensable before the contracts are released. It is tricky to detect vulnerabilities in smart contracts only through source codes, as source codes of most contracts are inaccessible. Clearly it is hard to manually check whether a contract without source codes is vulnerable and what types of vulnerabilities it has.

Manuscript received September 10, 2019; revised January 15, 2020; accepted January 15, 2020. Date of publication January 23, 2020; date of current version July 7, 2021. The work reported in this paper was supported in part by the Natural Science Foundation of China under Grant U1736114 and in part by the National Key R&D Program of China under Grant 2017YFB0802805. Chunhua Su was supported in part by JSPS Kiban(B) 18H03240 and in part by JSPS Kiban(C) 18K11298. Recommended for acceptance by Dr. Y. Wu. *(Corresponding authors: Yidong Li and Guangquan Xu.)*

W. Wang is with the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China and also with the Division of Computer, Electrical and Mathematical Sciences & Engineering (CEMSE), King Abdullah University of Science and Technology (KAUST), Thuwal 23955-6900, Saudi Arabia (e-mail: wangwei1@bjtu.edu.cn).

J. Song and Y. Li are with the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China (e-mail: 17120479@bjtu.edu.cn; ydli@bjtu.edu.cn).

G. Xu is with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, Tianjin 300350, China (e-mail: losin@tju.edu.cn).

H. Wang is with the Department of Computer Science, Norwegian University of Science and Technology, 2815 Gjøvik, Norway (e-mail: hawa@ntnu.no).

C. Su is with the Division of Computer Science, University of Aizu, Aizu-Wakamatsu 965-8580, Japan (e-mail: suchunhua@gmail.com).

Digital Object Identifier 10.1109/TNSE.2020.2968505

```
1  /* a lottery contract example in which players can particpate in
2     by sending tokens to the contract */
3  contract Lottery {
4      mapping(address => uint) usersBet;
5      mapping(uint => address) users;
6      uint nbUsers = 0;
7      uint totalBets = 0;
8      address owner;
9      // called by the constructor
10     function Lottery() {
11         owner = msg.sender;
12     }
13     // function that recieves tokens
14     function Bet() public payable  {
15         if (msg.value > 0) {
16             if (usersBet[msg.sender] == 0) {
17                 users[nbUsers] = msg.sender;
18                 nbUsers += 1;             }
19             usersBet[msg.sender] += msg.value;
20             totalBets += msg.value;
21         } }
22     // function for picking the winner and withdrawing tokens
23     function EndLottery() public {
24         if (msg.sender == owner) {
25             uint sum = 0;
26             uint winningNum = uint(block.blockhash(block.number-1)) % totalBets + 1;
27             for (uint i=0; i < nbUsers; i++) {
28                 sum += usersBet[users[i]];
29                 if (sum >= winningNum) {
30                     selfdestruct(users[i]);
31                     return;
32                 } } } }
33 }
```

Fig. 1. An Example of A Lottery Contract.

Fig. 1 shows an example of a lottery contract [17], written in Solidity language. The lottery smart contract described in Fig. 1 has the Timestamp Dependency vulnerability (explained in details in Section IV). In line 26, the variable $winningNum$ depends on the timestamp (*e.g.*, $BLOCKHASH$ & $NUMBER$) of the current block. In line 29, $winningNum$ is used as the judging condition for users' self-destruction. The miner can calculate the timestamp that is beneficial to her/him in advance, and set the timestamp to her/his own favorable time when mining, in order to delay or advance the user's self-destruction. If the miner accelerates the user's self-destruction, the cryptocurrency owned by the user will not be able to be sent out, and all the cryptocurrency the user holds will be frozen, resulting in pecuniary lose.Based upon mainstream vulnerability detection methods, such as symbolic execution, formal verification with F* framework and K framework, and symbolic analysis, some tools have been publicly released. Comparing the detection time among these tools, Oyente [18], Mythril [19] and Securify [20] requires around 28 seconds, 60 seconds and 18 seconds, respectively, to detect vulnerabilities in each contract. We see that prior tools are time-consuming and may not be suitable for batch vulnerability detection, as these tools mainly employ symbolic execution or symbolic analysis that requires the exploration of all executable paths in a contract or analysis of dependency graphs of the contract.

In this work, we propose ContractWard in the aim to improve the efficiency of vulnerabilities detection in smart contracts on the premise of ensuring accuracy of the detection based on machine learning techniques. It is able to effectively and fast detect vulnerabilities based on their patterns learned from training samples. We build ContractWard in three steps. First, we collect 49502 verified Ethereum smart contracts with source codes written in Solidity language [3] in November 2018. We then label the contracts as six types of vulnerabilities with Oyente. Second, we extract typical features that describe static characteristics of contracts from operation codes (opcodes) [21], since source codes of most contracts are inaccessible. Source codes are complied into bytecodes [22] and bytecodes are translated into operation codes (opcodes). Finally, we employ machine learning algorithms to detect vulnerabilities in smart contracts. We adopt two sampling algorithms, namely, Synthetic Minority Oversampling Technique (SMOTE) [23] and SMOTE-Tomek [24], to balance the training data sets, as the data is class-imbalance. We then employ five machine learning algorithms, namely, eXtreme Gradient Boosting (XGBoost) [25], adaptive boosting (AdaBoost) [26], Random Forest (RF) [27], Support Vector Machine (SVM) [28] and k-Nearest Neighbor (KNN) [29] to detect whether a test smart contract is vulnerable or not.

We make the following contributions:

- We propose a system called ContractWard for large-scale and automated vulnerability detection on Ethereum smart contracts with machine learning algorithms. Different from the existing work that mainly relies on symbolic execution, ContractWard learns the patterns of vulnerable contracts in training samples for the detection.
- To better characterize features of smart contracts, we collect 49502 real-world smart contracts from Ethereum official website [30]. We further extract 1619 dimensional bigram features from simplified operation codes to construct a feature space.
- ContractWard detects six vulnerabilities of smart contracts quickly, effectively and automatically. We run ContractWard on real contracts and the predictive recall and precision of the system reach over 96%. In addition, its detection time is about 4 seconds per contract. ContractWard is proven to be time-saving and suitable for batch detection of vulnerabilities in smart contracts.

The remainder of this paper is organized as follows. Related work is summarized in Section II. We then briefly introduce some concepts about Ethereum and smart contracts in Section III. Next we review six well-known vulnerabilities in Section IV. Section V provides the description of the data, features and models. Experiments and results are presented in Section VI. Discussion and analysis are provided in Section VII. Section VIII concludes this paper.

II. Related Work

The security issues of smart contracts have drawn public attentions [16]. The work on formal verification can be traced back to 2016 when Hirai [31] used the Isabelle proof assistant to formally verify contracts. Grishchenko *et al.* [32] defined formal semantics for contract source codes with $F*$ framework, and Hildenbrandt *et al.* [33] defined it with K framework. These semantics were executable. However, they are not fully automated as they only focused on arbitrary properties. Then Tsankov *et al.* [20] proposed an automated security tool called Securify. Bhargavan *et al.* [33] provided a strategy to verify contracts through putting source codes and bytecodes into an existing verification system. However, they did not measure

their tool on real-world contracts. The prototype of ZEUS [34] claimed that it was a sound analyzer with zero false negatives by translating contracts into LLVM framework. Jiang *et al.* [35] provided ContractFuzzer to test Ethereum smart contracts whether they are vulnerable or not. Due to random generation of test Oracle, the system could not cover all paths and thus it was difficult to find all potential vulnerabilities.

Based upon symbolic execution, there are some popular automated security tools for contracts, including Oyente [18], Maian [36] and Mythril [19]. The main idea of symbolic execution was to replace arbitrary uncertain variables in source codes, such as environmental variables and formal parameters, with symbolic values in the process of analysis. Symbolic execution is a powerful generic method to detect vulnerabilities. However, it may not cover all execution paths, resulting in false negatives. In addition, it is time-consuming for exploration of all the executable paths. In our previous work [37], we proposed an efficient vulnerability detection model for smart contract. In this work, we extend our previous work by proposing ContractWard that is an automated vulnerability detection model based on machine learning techniques. Compared with these tools, ContractWard learns the patterns of vulnerable contracts in training samples. It does not need to repeatedly execute the same opcodes, and thus can be time-saving. In our previous work, we detected anomalies or malware with static [38]–[42] or dynamic analysis [43]–[45] or with network traffic [46]–[49]. While our previous work detected potential malicious behaviors in systems or networks, in this work, we mainly focused on the detection of vulnerabilities in the smart contracts running on blockchain.

III. Background

We briefly introduce Ethereum and smart contracts in this section. The discussion of smart contracts is confined to Ethereum while the language for contract source codes is restricted to Solidity language.

A. Ethereum in a Nutshell

Ethereum is the most popular second-generation blockchain platform that features smart contracts. In essence, Ethereum is a distributed (also decentralized) ledger that takes blockchain as its basic support technology like Bitcoin. It supports execution and invocation environment of smart contracts through a Turing-complete machine that is called Ethereum Virtual Machine (EVM) [50]. EVM makes it easy for developer to construct decentralized applications on Ethereum. It thus greatly expands application scope on blockchain. EVM explains how to change system state given a series of instructions and a small part of environmental data [51]. The word size of the EVM (hence the size of the stack item) is 256 bits, which facilitates KECCAK256 hash scheme and elliptic curve calculation [51]. EVM has a simple stack structure with a maximum stack size of 1024. If call count is over 1024, the Callstack Attack may take place.

From the theoretical standpoint, Ethereum is bound to face a problem of consistency due to its distributed characteristics, and the solution is consensus [51]. At present, the proof of work (PoW) algorithm as a consensus is used on Ethereum. When a contract receives a message call, its codes will be executed on EVM of each node simultaneously, and finally the consensus is reached on the execution results. The main idea of the consensus is to encourage nodes to pay contribution to the system with economic incentives, and to prevent nodes from doing evil with economic punishment. In order to embolden more nodes to participate in consensus, the system rewards cryptocurrency to the contributing nodes.

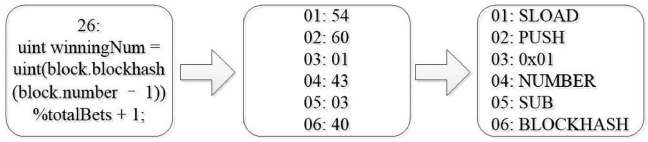


Fig. 2. The Relationships among Source Codes, Bytecodes and Opcodes.

B. The Bytecodes and Operation Codes of Smart Contracts

On EVM, it works with three steps to deploy a contract [25]. First, source codes are written in high-level language (*e.g.*, *Solidity*) by developers. Second, the source codes are compiled into bytecodes with a compiler. Bytecodes (or called EVM code) are byte arrays encoded by hexadecimal digits. Finally, the bytecodes are uploaded to EVM with an Ethereum client. On basis of one byte, the bytecodes can be translated into EVM instructions or operation codes (opcodes). In accordance with Ethereum Yellow Paper, there are 135 operation instructions with 10 functions, namely, stop and arithmetic operations, comparison and bit-wise logic operations, SHA3 operations, environment information operations, block information operations, stack, memory, storage and flow operations, push operations, exchange operations, logging operations and system operations [50].

Currently some instructions are not defined and they will be defined only for future expansion. As a large number of man-made variables are defined in source codes, analyzing smart contracts with source codes may not be appropriate. For example, there are two contracts named A and B where the function declaration of contract A is '$function\ transfer(address\ \_to,\ uint256\ \_value)$,' while the function declaration of contract B is '$function\ transfer(address\ \_receiver,\ uint256\ \_token)$'. They look quite different from source codes, but are similar in opcodes. Clearly it would be easier to analyze smart contract with opcodes. Fig. 2 illustrates the relationships among source codes, bytecodes and opcodes.

IV. Six Types of Security Vulnerabilities in Smart Contracts

A. Integer Overflow and Integer Underflow Vulnerabilities

The value of integer type in computer language has a range of maximum (max) and minimum (min) number. The integer type is unsigned on blockchain and thus the minimum is 0. Suppose a scenario where unsigned integer is 8 bits, the maximum is thus 2^8. Integer Overflow and Integer Underflow

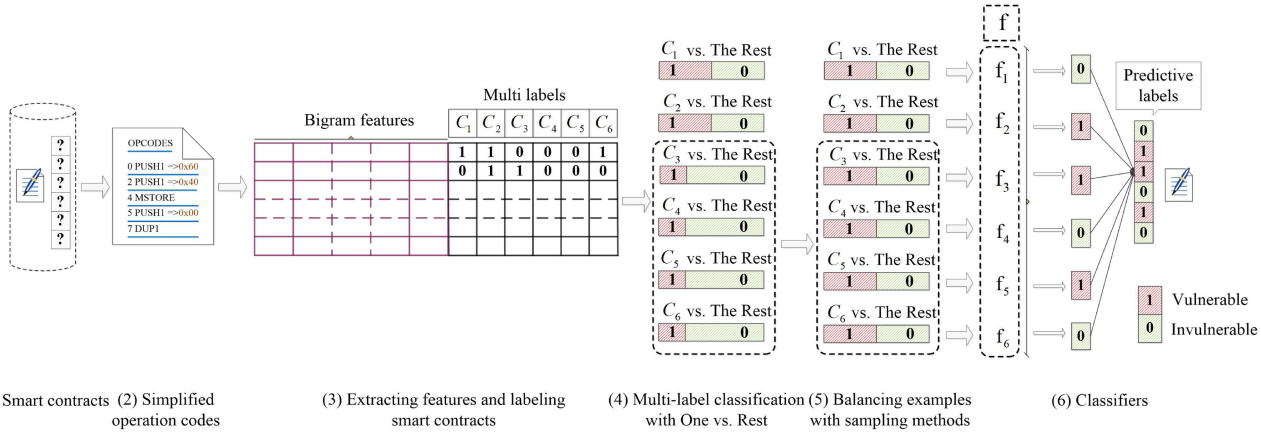


Fig. 3. The Process of Training Models.

Vulnerabilities arise when calculation exceeds max or is below min because of $max + 1 \rightarrow min$ or $min - 1 \rightarrow max$.

B. Transaction-Ordering Dependence (TOD)

On blockchain, the performance of smart contracts varies with different transaction sequences. Unfortunately, the sequences may be manipulated by miners. Consider a case where the pending transaction pool (txpool for short) has two new transactions ($e.g.$, T; T_i) and the blockchain is at state S, and the state S can be transformed into state S_1 only if the transaction T is handled. Originally, T should be processed at state S, and thus the state is from S into S_1. But miners can deal with the transaction T_i prior to T in accordance with their own wishes, and then the state is from S into S_2 rather than from S into S_1. Therefore if T is processed at this time, the state will be changed into another new state S_3. In the aforementioned case, T is processed in different block states and vulnerabilities result from the changes of the expected transaction sequences.

C. Callstack Depth Attack Vulnerability

On Ethereum, a contract can invoke other contracts via some instructions, such as $.send()$, $.call()$, $.delegatecall()$ and $.transfer()$. Nevertheless, if the depth of call stack goes beyond the threshold ($e.g.$, 1024), the instructions will not throw an exception but return false except $.transfer()$. If the return values are unchecked, the caller does not realize the failure of calling. Therefore, contracts should check the return values of instructions to determine whether the execution is on schedule.

D. Timestamp Dependency

This vulnerability happens when a contract uses block variables as a call condition to perform some critical operations ($e.g.$, $sending\ tokens$) or as a seed to generate random numbers. Some variables root in block header, including BLOCKHASH, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT and COINBASE, and therefore, in principle, they can be effected by miners. For instance, miners have right to set the block TIMESTAMP within 900 seconds offset. If cryptocurrency is transferred based on block variables, miners can exploit the vulnerability by tampering with them.

E. Reentrancy Vulnerability

Reentrancy vulnerability is a notorious vulnerability. Smart contracts feature to invoke and utilize codes from external contracts. The performance of triggering an external contract or sending cryptocurrency to an account requires to submit an external call. The external call may be hijacked by an attacker to force the contracts to execute reentrant codes including calling back themselves. Therefore, the same codes are executed repeatedly like the indirect recursive function calls in the programming language. The vulnerability was discovered in the DAO contract in 2016.

V. Detection Models

As shown in Fig. 3, ContractWard is built with six steps. First, we collect a big number of fresh and verified smart contracts from the official website of Ethereum. Second, source codes are transformed into operation codes (opcodes) (as described in Fig. 2). Then opcodes are simplified. Third, we extract 1619 dimensional bigram features from simplified contract opcodes and label contracts with six types of vulnerabilities (discussed in Section IV). Fourth, we employ One vs. Rest (OvR) algorithms for multi-label classification where C_1, C_2, C_3, C_4, C_5 and C_6 correspond to Integer Overflow vulnerability (Overflow), Integer Underflow vulnerability (Underflow), Transaction-Ordering Dependence (TOD), Callstack Depth Attack vulnerability (Calltack), Timestamp Dependency (Timestamp) and Reentrancy vulnerability (Reentrancy). Fifth, for balanced examples like C_1 vs. the rest or C_2 vs. we perform the classification directly. For the remaining four types of vulnerabilities, we need to employ sampling algorithms to balance them before classification because of class imbalance [52]. Finally, we build detection models on the balanced training sets for the detection.

A. Data Sets, Labels, and Feature Space

1) Data Sets: We collect 49502 smart contracts with source codes from the official website of Ethereum, where smart contracts has been verified before September 2018. The data is

TABLE I
THE DESCRIPTION OF DATA SETS

Type	Number		
	This Type (Vulnerable)	The Rest (Invulnerable)	Total
Overflow	22128	27374	49502
Underflow	9699	39803	49502
TOD	1436	48066	49502
Callstack	192	49310	49502
Timestamp	477	49025	49502
Reentrancy	100	49402	49502

TABLE II
THE SIMPLIFICATION RULES

Substituted Opcodes	Original Opcodes
ARITHMETIC_OP	ADD MUL SUB DIV SDIV SMOD MOD ADDMOD MULMOD EXP
CONSTANT1	BLOCKHASH TIMESTAMP NUMBER DIFFICULTY GASLIMIT COINBASE
CONSTANT2	ADDRESS ORIGIN CALLER
COMPARISON	LT GT SLT SGT
LOGIC_OP	AND OR XOR NOT
DUP	DUP1-DUP16
SWAP	SWAP1-SWAP16
PUSH	PUSH5-PUSH32
LOG	LOG1-LOG4

clearly reliable, authoritative and comprehensible. The data sets contain contracts with the six types of vulnerabilities. The description of data sets are shown in Table I. For Integer Overflow vulnerability and Integer Underflow vulnerability, the ratios of negative (invulnerable) to positive (vulnerable) examples are balanced. For remaining four types of vulnerabilities, the ratios of negative to positive examples are quite imbalanced, even up to 100:1, where negative examples are in the majority class and positive examples are in the minority class without exception. In general, if the ratio of one category to another is over 5:1, the examples are considered to be class-imbalance.

2) Labels: We employ Oyente [18] to label all the contracts, and each contract is with six labels. Then we manually check the correctness of the labels. The labels are independent from each other in each type of vulnerability. For instance, an example with the multi-label vector like [1 0 1 0 0 0] demonstrates that it has the first and the third vulnerabilities and an example with [0 0 0 0 0 0] has no vulnerability, theoretically.

Oyente has been updated in July 2018 [53], including but not limited to (1) reducing false positives of Reentrancy vulnerability by adding a threshold to the sending amount ($e.g.,\ sending\ gas > 2300\ and\ sending\ tokens > depositing\ tokens$); (2) newly adding Callstack vulnerability, Integer Overflow vulnerability and Integer Underflow vulnerability detection; and (3) considering a revertible overflow as a false positive of Overflow vulnerability. Meanwhile, many papers ($e.g.$, [21], [34], [35]) have used Oyente as a benchmark for comparison. We assume that the labels generated by Oyente are reliable.

TABLE III
THE BIGRAMS EXTRACTED FROM OPCODES

A Simplified Opcodes Fragment	The Bigrams
PUSH1, CALLDATALOAD, PUSH, SWAP, ARITHMETIC, PUSH4, LOGIC_OP	_, PUSH1; PUSH1, CALLDATALOAD; CALLDATALOAD, PUSH; PUSH, SWAP; SWAP, ARITHMETIC; ARITHMETIC, PUSH4; PUSH4, LOGIC_OP; LOGIC_OP, _;

3) Feature Space: We employ n-gram algorithm [54] for feature extraction. N-Gram refers to n words that appear continuously in text. It is a probabilistic language model in view of first-order Markov Chain hypothesis where words are only related to those few in front of them and thus there is no need to trace back to the first opcode in smart contracts. Through a sliding window of binary-byte size, opcodes are segmented into massive n-grams [54]. In particular, unigrams, bigrams and trigrams are examples of n-gram, where n is 1, 2 and 3, respectively. In other words, the next word appears depending on the word before it, which is called bigram, while the next word appears depending on the two words that precede, which is called trigram. We use bigrams as features in this work. According to our statistics, the opcode length of each contract is about 4364 on average and there are more than 100 types of opcodes in total. Therefore directly using n-gram algorithm to extract features may lead to the curse of dimensionality caused by excessive number of features.

In order to reduce the dimensionality of the features, we simplify opcodes by dislodging the operands and classifying functionally similar opcodes into one category. In details, each push instruction is followed by an operand, which can be removed. For block information instructions, a simplified opcode is acted as the substitution for six opcodes, which have the same impact on Timestamp vulnerability. Thus after the processing, there are only about 50 opcodes remained. The simplified rules of opcodes are described in Table II.

As shown in Table III, after simplification, we extract bigram features from a simplified opcodes fragment. Each distinct bigram is a feature and ultimately we extract 1619 dimensional features, which are used to identify vulnerabilities. We construct a feature space (FS) where each contract has its corresponding feature vector. Each feature value in a feature vector is calculated as the ratio of the number of each bigram to the number of bigrams occurred in the contract. The feature space (FS) is defined in Equation (1).

$$FS = \begin{pmatrix} f_{11} & f_{12} & \cdots & \cdots & f_{1,1619} \\ f_{21} & f_{22} & \cdots & \cdots & f_{2,1619} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ f_{i1} & \cdots & f_{ij} & \cdots & f_{i,1619} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \tag{1}$$

where f_{ij} is the feature frequency of the ith bigram in the jth contract, a decimal number between 0 and 1. We define n_{i,c_j} as the number of the ith bigram occurrences in the jth contract and define n_{all,c_j} as the sum of all bigram occurrences in the same contract. Then we define f_{ij} in Equation (2) as

$$f_{ij} = \frac{n_{i,c_j}}{n_{all,c_j}} \tag{2}$$

Note that if a bigram feature does not appear in the contract, the corresponding f_{ij} is 0.

B. Training Sets

In general, training sets can include up to 70% samples randomly selected from the whole data sets in supervised classification. However, in this work, as mentioned, the training sets are imbalanced because classification categories are not roughly equally represented. As an example, a typical imbalanced data sets might contain 97% invulnerable examples and 3% vulnerable examples. If all the examples are identified as invulnerable examples (the majority class), the accuracy of the guessing would be 97%. However, the vulnerability detection aims to gain high recall (R) and precision (P) in vulnerable examples (the minority class). For the imbalanced data sets, only high accuracy is obviously not appropriate.

In our training sets, for some types of vulnerabilities, the ratios between negative and positive examples are quite imbalanced, even up to 100:1. To deal with this problem, we take a measure to diminish the class-imbalance impact in the training sets. In details, we adopt Synthetic Minority Oversampling Technique (SMOTE) [23] and the combination of SMOTE and TomekLinks (SMOTETomek for short) [55] to extend the number of minority class to be similar to that of the majority class. SMOTE is an oversampling technique, interpolating between the minority class to generate extra ones. When SMOTE algorithm is used, samples with invalid information may be generated, thus increasing the overlap in the minority class. However, SMOTETomek is a combined sampling technique. Its oversampling using SMOTE is followed by undersampling that can remove samples with neighborhood relations (Tomek's links). Therefore it can delete useless samples during sampling. Both of them support multi-label resampling. The ratio of original training sets, the ratio of training sets balanced with SMOTE and the ratio of training sets balanced with SMOTETomek among categories are described in Table IV. We employ five supervised learning algorithms based on the balanced data sets to achieve multi-label classification.

TABLE IV
THE RATIOS OF POSITIVE TO NEGATIVE SAMPLES AFTER SAMPLING

Categories	Origin	SMOTE	SMOTETomek
TOD vs.The Rest	3:100	1:1	1:1
Callstack vs. The Rest	1:250	1:1	1:1
Timestamp vs. The Rest	1:100	1:1	1:1
Reentrancy vs. The Rest	1:500	1:1	1:1

C. Classification Algorithms

In our training sets where $y_i \in \{C_1, C_2, C_3, C_4, C_5, C_6\}$ and $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_i, y_i), \ldots, (x_k, y_k)\}$, the multi-label classification task is realized by splitting, that is, dividing a multi-label classification task into several binary classification tasks. A classifier is trained for each binary classification task and finally six classifiers are trained. The classification results of these binary classifiers are integrated to provide the final results of multi-label classification. We employ One vs. Rest (OvR) strategy, one of the most classic splitting strategies, to fulfill multi-label classification. The main idea of OvR is to train six binary classifiers in the condition of taking one category as positive class and the other categories as negative class. In the process of training, if a sample is predicted as positive in some of the six categories, then the corresponding labels would be 1, which means the sample has vulnerabilities in these categories.

Ensemble learning algorithms are employed in each binary classification task to obtain better generalization performance than a single learner. To facilitate comparison, single learning algorithms are also employed. The ensemble learning algorithms accomplish learning tasks by combining multiple base learners and we use Decision Tree (DT) in this work. In general, the ensemble learning algorithms are of two kinds on the whole: Boosting and Bagging, where the former is a serialization method with strong dependence among base learners that have to be generated serially, while the latter is a parallelization method with the weak dependence among learners that can be generated simultaneously. From the perspective of bias-variance, Boosting aims to reduce bias, so it can set up a strong ensemble learner on the base learners with relatively weak generalization ability.

Based on the feature space and labels of training sets, we employ eXtreme Gradient Boosting (XGBoost) to develop ContractWard to detect vulnerabilities in smart contracts. We also adopt Adaptive Boosting (AdaBoost), Random Forest (RF), Support Vector Machine (SVM) and k-Nearest Neighbor (KNN) for the detection for comparison.

- eXtreme Gradient Boosting (XGBoost): XGBoost is an efficient Boosting algorithm. In order to realize fast fit, the learner should minimize the difference between predicted values and factual values ($e.g.,\ residual\ errors$), and form the regularized loss function. Finally, the prediction is the summation of all the learners. XGBoost draws on the column sampling similar to RF to reduce the variance. Compared to AdaBoost, XGBoost is efficient as it supports parallel processing at feature granularity instead of the learner granularity.
- Adaptive Boosting (AdaBoost): AdaBoost is a representative of Boosting algorithms. It starts establishing its first learner with the initial training sets. In the process of re-weighting, it increases the weights of the samples that are correctly classified or predicted while reduces the weights of the samples that are incorrectly classified and predicted. After the weights of the

samples are redistributed, the next weak learner training is carried out [56]. This is repeated until the number of base learners reaches the pre-set value. Finally, the strong learner can be obtained by combining multiple weak learners. After the training process of each weak classifier, the weight of the weak classifier with small classification error rate is increased, so that it plays a greater decisive role in the final classification functions. In contrast, the weight of the weak classifier with large classification error rate is reduced, so that it plays a smaller decisive role in the final classification functions.

- Random Forest (RF): RF is an extended variant of Bagging algorithms. The training sets are composed of n examples using random sampling algorithm with replacement to sample n times from the data sets, repeatedly, until t number of training sets are obtained. And then t base learners are trained, respectively. Next, in the process of prediction, classification decision depends on a majority vote. Random attribute selection is used in the process of training.
- Support Vector Machine (SVM): SVM is the widely used classification method. Its goal is to find a hyperplane to segment samples into positive or negative samples, so there is a maximum margin between the two categories, where the classifier has high reliability and good generalization ability for new samples.
- k-Nearest Neighbr (KNN): KNN is also very widely used classification algorithm. It is simple but efficient. Given a test sample, k training samples closest to the sample are found based on some distance measure, and then the prediction is gained on the information of the k neighbors. According to majority vote, the most frequent category labels in k samples are selected as the prediction results.

D. Model Selection

The classification results vary greatly with hyper-parameters for the same learning algorithm. Hyper-parameters are parameters whose values should be set before learning, instead of parameters that can be obtained through training. Therefore, in the model selection, hyper-parameters of algorithms should be adjusted, which is commonly called hyper-parameters tuning. The models are trained with pre-set hyper-parameters, and then the hyper-parameters of the optimal model are obtained by parameter tuning.

In addition, the decision threshold is also adjusted in accordance with data distribution called threshold moving. In general, if the prediction value is over the threshold with 0.5 as default, the sample is discriminated as positive, or it is negative, conversely. Through above methods, the problem of over-fitting and under-fitting in classifiers can be well averted in the training process. In our work, we do not employ the n-fold cross-validation in training processing.

VI. Evaluation

In this section, we conduct comprehensive experiments on the test sets to achieve triple targets. First, we compare the sampling methods with five classifiers to verify the necessity of sampling methods. Second, we use F1-score, Micro-F1 value, Macro-F1 value to measure the performance of the classifiers. Based on the evaluation results, we use XGBoost classifier trained on the balanced training sets with SMOTETomek in our ContractWard. Finally we analyze the classification results of ContractWard in details.

TABLE V
The Experiment Environment

Software and Hardware	Configurations
Server Model	Lenovo ThinkServer RD640
Operating System	Ubuntu 18.04
CPU	Xenon E5-2680 v2
Memory Size	320GB
Disk Capacity	3.3TB

A. Experimental Setup

The experiments are preformed based on a large number of data sets, and thus have high requirements for CPU performance, hard disk capacity and memory size of experimental device. Our experiment environment is described in Table V.

B. Test Sets

In the experiments, 70% of data sets are used as training data. If the remaining 30% is used as test data directly, the results of classification may not be good enough on imbalanced test sets. It is essential to balance the test sets without incorporating the authenticity of samples.

As mentioned, in order to balance the test sets, we adopt random sampling method to select samples from around $15K$ real-world smart contracts. For four types of vulnerabilities, namely, TOD, Callstack vulnerability, Timestamp vulnerability and Reentrancy vulnerability, we randomly select samples from the majority class, and the number of samples that are selected from the majority class is five times as many as the number of the minority class. We then combine the samples selected from the majority class and all the samples of the minority class to form the test sets that finally contain enough samples without fictitious samples.

C. The Comparison of Sampling Methods

We evaluate our methods on real-world smart contracts. We use five classifiers, namely, XGBoost classifier, AdaBoost classifier, RF classifier, SVM classifier and KNN classifier on the same test sets and each classifier is trained by three distinct training sets, namely, original training sets, training sets balanced with SMOTE and training sets balanced with SMOTETomek as shown in Table IV. We choose Micro-F1 and Macro-F1 as evaluation indices of classification. Micro-F1 and Macro-F1 are measurements used to evaluate multi-label classification. When calculating Micro-F1, the value is susceptible to the classification results of categories with many samples. When calculating Macro-F1, the weights of each category are equal regardless of the number of samples in each category. Micro-F1

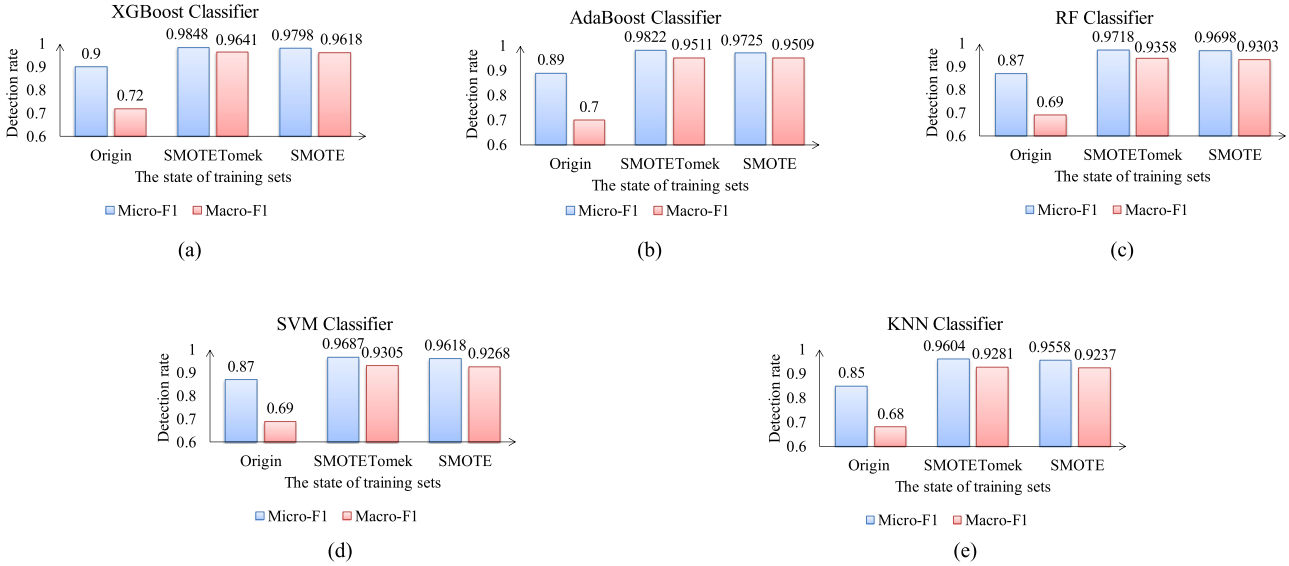


Fig. 4. The Comparison of Sampling Methods with Five Classifiers.

and Macro-F1 are defined in the following Equations:

$$macro-P=\frac{1}{n}\sum_{i=1}^{n}P_i,\ macro-R=\frac{1}{n}\sum_{i=1}^{n}R_i \tag{3}$$

$$macro-F1=\frac{2\times macro-P\times macro-R}{macro-P+macro-R} \tag{4}$$

$$micro-P=\frac{\overline{TP}}{\overline{TP}+\overline{FP}},\ micro-R=\frac{\overline{TP}}{\overline{TP}+\overline{FN}} \tag{5}$$

$$micro-F1=\frac{2\times micro-P\times micro-R}{micro-P+micro-R} \tag{6}$$

where True Positives (TP) is the number of positive samples correctly classified as positive samples; True Negatives (TN) is the number of negative samples correctly classified negative samples; False Positives (FP) is the number of negative samples incorrectly classified as positive samples and False Negatives (FN) is the number of positive samples incorrectly classified as negative samples. $\overline{TP}$, $\overline{FP}$ and $\overline{FN}$ are the mean values of TP, FP and FN, respectively.

Fig. 4 demonstrates that the predictive Micro-F1 and Macro-F1 values of each classifier trained on training sets balanced with SMOTE or on training sets balanced with SMOTETomek are larger than those of each classifier trained on the original training sets. More concretely, SMOTETomek is more efficient than SMOTE in five classifiers to balance our data and Micro-F1 and Macro-F1 can both reach over 96% within XGBoost classifier. Therefore SMOTE and SMOTETomek methods can successfully solve the problem of weak generalization ability of classifiers caused by class imbalance.

D. The Comparison of Classifiers

We conduct comprehensive experiments in the aim to compare the performance based on the five multi-label classifiers, namely, XGBoost classifier, AdaBoost classifier, RF classifier, SVM classifier and KNN classifier, together with two sampling methods, namely, SMOTE and SMTOETomek. F1-score, Micro-F1, Macro-F1 are used to measure the performance of the classifiers. F1-score is a measure used to evaluate binary classifiers and it is defined as a weighted harmonic mean of recall (R) and precision (P). Compared with geometric mean (G-mean), it attaches more importance to the positive class and is defined in Equation (7) as

$$F1-score=\frac{2\times P\times R}{P+R} \tag{7}$$

In Table VI, it is seen that XGBoost classifier produces higher F1-score values than AdaBoost classifier, RF classifier, SVM classifier and KNN classifier in each binary classification task. The predictive Micro-F1 value and Macro-F1 value of XGBoost multi-label classifier are the highest among the five classifiers, and they all reach over 96%. Ensemble learning classifiers perform better than SVM and KNN classifiers in our multi-label classification. Micro-F1 value is larger than Macro-F1 value, because the number of test samples for both Overflow and Underflow Vulnerabilities are large and F1-score values to these two categories are high. Comparing Table VI and Table VII, it is clear that the performance of XGBoost classifier trained by training sets balanced with SMOTETomek is better as expected. Thus we choose XGBoost classifier together with SMOTETomek method in our models called ContractWard.

E. The Analysis of Our ContractWard

1) The ROC Curves: Receiver Operating Characteristic (ROC) curves are used to measure the performance of ContractWard by trading off relative costs of True Positive Rate (TPR) and False Positive Rate (FPR) where $TPR=TP/(TP+FN)$ and $FPR=FP/(TN+FP)$. On ROC curves,

TABLE VI
THE DETECTION PERFORMANCE COMPARISON OF FIVE CLASSIFIERS TRAINED BY TRAINING SETS BALANCED WITH SMOTETOMEK

Classifiers	F1-score						Micro-F1	Macro-F1
	Overflow	Underflow	TOD	Callstack	Timestamp	Reentrancy		
XGBoost	**0.99**	**0.99**	**0.96**	**0.97**	**0.95**	**0.95**	**0.9848**	**0.9641**
AdaBoost	0.98	0.98	0.95	0.93	0.93	0.94	0.9822	0.9511
RF	0.98	0.98	0.93	0.91	0.91	0.92	0.9718	0.9358
SVM	0.98	0.97	0.92	0.91	0.91	0.92	0.9687	0.9305
KNN	0.97	0.97	0.92	0.91	0.91	0.91	0.9604	0.9281

TABLE VII
THE DETECTION PERFORMANCE COMPARISON OF FIVE CLASSIFIERS TRAINED BY TRAINING SETS BALANCED WITH SMOTE

Classifiers	F1-score						Micro-F1	Macro-F1
	Overflow	Underflow	TOD	Callstack	Timestamp	Reentrancy		
XGBoost	**0.99**	**0.99**	**0.97**	**0.97**	**0.95**	**0.94**	**0.9798**	**0.9618**
AdaBoost	0.98	0.98	0.95	0.93	0.93	0.94	0.9725	0.9509
RF	0.98	0.98	0.94	0.91	0.92	0.90	0.9698	0.9303
SVM	0.97	0.97	0.92	0.92	0.91	0.91	0.9618	0.9268
KNN	0.97	0.97	0.92	0.90	0.91	0.91	0.9558	0.9237

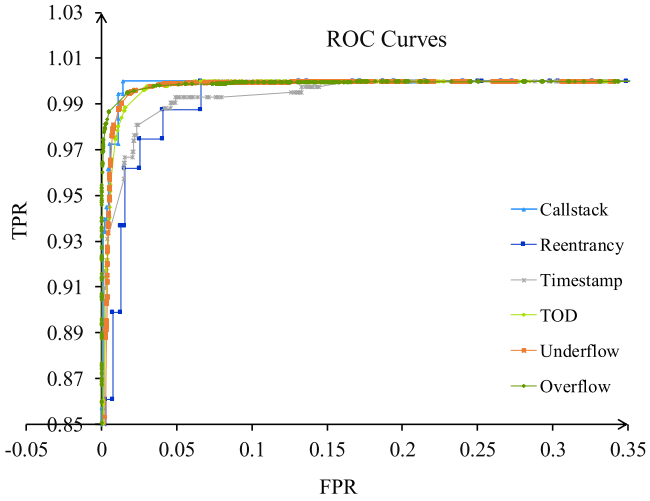


Fig. 5. The ROC Curves of Contractward with XGBoost classifier.

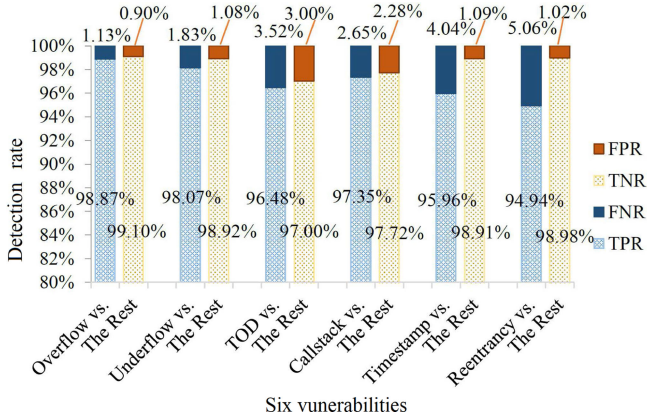


Fig. 6. TPRs, FNRs, TNRs, and FPRs of Contractward.

FPR is represented as X-axis and TPR is represented as Y-axis. Theoretically, the ideal point would be (0,1), which means all positive samples and negative samples are classified correctly, respectively. Thus the point closer to top left indicates that classification results are better. The ROC curve of ContractWard with XGBoost classifier is shown in Fig. 5.

2) TPRs, FNRs, TNRs and FPRs: As presented in Fig. 6, $Overflow\ vs.\ The\ Rest$ means $The\ Rest$ is invulnerable and $Overflow$ is short for Integer Overflow vulnerability. All the TPRs are over 94% and the TNRs are over 97% for each type of vulnerability. High TPR and TNR demonstrate that the effectiveness of ContractWard in terms of classification.

In particular, ContractWard is successfully detecting the vulnerabilities of BEC contracts and DAO contracts that have resulted in huge economic losses.

3) The Comparison in Detection Time: As mentioned, ContractWard takes three steps to detect the six types of vulnerabilities of smart contracts: (1) compiling contracts from source codes into bytecodes and then translating the bytecodes into opcodes (C& T for short); (2) constructing 1619 dimensional bigram features from each contract; (3) predicting whether there are vulnerable and what kinds of vulnerabilities in the contracts. In the experiments, approximate $3K$ real-world smart contracts are detected by ContractWard. According to the statistics, ContractWard spends about 4 seconds in detecting a contract, which is much faster than Oyente and Securify. Oyente takes about 28 seconds and Securify takes about 18 seconds to detect vulnerabilities on per contract. The comparison of vulnerability detection time among them is shown in Table VIII.

VII. ANALYSIS AND DISCUSSION

We extract 1619 dimensional bigram features from each smart contract. These features are independent with high-level programming languages for writing smart contracts on Ethereum. Therefore, they can effectively describe the static

TABLE VIII
THE COMPARISON IN DETECTION TIME (SECONDS)

	ContractWard			Oyente	Securify
	C&T	Extracting	Predicting		
Times	3.90	0.02	0.15	28.50	18.40

characteristics of contracts. We also extract unigram features and trigram features for comparison. As an unigram feature only contains one operation instruction, it does not reflect the connection between instructions. As a result, the predictive precision and recall of the model based on unigram features is lower than those of the models based on bigram features. In contrast, based on trigram features, the classification results of the model are not better than the model based on bigram features. At the same time, the model based on trigram features are inefficient, because excessive features increase the difficulty of constructing the model and manual tuning of hyperparameters. Therefore, based on bigram features, ContractWard is time-saving. It also has high precision and recall in vulnerability detection in smart contracts.

ContractWard is very efficient in the detection of vulnerabilities in smart contracts. First, we simplify opcodes. Therefore, the number of features extracted with n-gram (n = 2) is decreased. In other words, the input data of ContractWard is simplified. Second, the essence of the supervised machine learning algorithm is to train an objective function, which can describe the mapping relationships between feature space and the labels of samples. In the process of training, ContractWard acquires the parameters of the objective function through progressive iteration and update. In the process of prediction, ContractWard can thus directly predict whether a new sample is vulnerable or not and what types of vulnerabilities it belongs to with the parameters learned during the training. There are two typical tools for detecting vulnerabilities in smart contracts. Oyente is a tool to detect contract vulnerabilities based on symbol execution. In the process of detection, it requires the exploration of all executable paths in a contract. Meanwhile, loop body needs to be iterated. It is thus time-consuming. Securify is another tool to detect vulnerabilities. It extracts precise semantic facts by symbolically analyzing dependency graphs of contracts, and uses these facts to match compliance and violation patterns. It is also time-consuming in constructing dependency graphs and matching patterns.

The accuracy of vulnerability detection with ContractWard depends on the authenticity of labels generated by Oyente. Oyente has few false positives for vulnerability detection, as it has made a series of updates as described in Section V. We also manually check the labels to ensure the correctness of the labeling information. ContractWard is based on the training of existing knowledge in vulnerable contrasts. Therefore, the vulnerabilities that have not been learned, or undefined new vulnerabilities cannot be recognized with ContractWard.

VIII. CONCLUSION

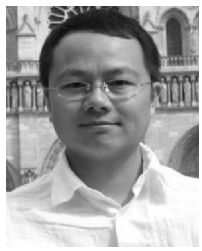
To secure the contract layer on Ethereum and purify Dapps markets, in this work, we propose ContractWard that is a model for effectively and efficiently detecting six types of vulnerabilities of smart contracts based on extracted static characteristics. We employ three supervised ensemble classification algorithms, namely, XGBoost, AdaBoost and RF, and two simple classification algorithms, namely, SVM and KNN, together with two sampling methods, namely, SMOTETomek and SMOTE to conduct comparative experiments. Finally, we select the model that takes XGBoost as the multi-label classifier and takes SMOTETomke as the sampling method in our ContractWard. The experimental results demonstrate the effectiveness and efficiency of ContractWard. First, the bigram features extracted from simplified opcodes systematically represent static features of contracts. Second, ContractWard is appropriate for rapid batch detection of vulnerabilities in smart contracts, and its detection speed is about 4 seconds on a smart contract on average, much more quickly than Oyente and Securify. Finally, ContractWard is reliable with the predictive Micro-F1 and Macro-F1 over 96%. ContractWard can be applied to detect vulnerabilities in smart contracts written in all high-level languages such as Solidity, Serpent and LLL, because high-level languages can all be converted into opcodes.

In future work, in order to improve the performance of ContractWard, we will explore more effective features to describe the characteristics of smart contracts. Designing anomaly detection models to detect novel vulnerabilities in smart contracts is also being investigated.

REFERENCES

[1] N. Szabo, "Smart contracts: Building blocks for digital markets," 1996.
[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[3] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, 1st ed., Apress, 2017.
[4] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," White Paper, vol. 3, p. 37, 2014.
[5] Z. Wan, Z. Guan, and X. Cheng, "Pride: A private and decentralized usage-based insurance using blockchain," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, 2018, pp. 1349–1354.
[6] Y. Zhang *et al.*, "Distributed electrical energy systems: Needs, concepts, approaches and vision," *Acta Automatica Sinica*, vol. 43, no. 9, pp. 1544–1554, 2017.
[7] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.
[8] A. Ekblaw, A. Azaria, J. D. Halamka, and A. Lippman, "A case study for blockchain in healthcare: "MEDREC" prototype for electronic health records and medical research data," in *Proc. IEEE Open Big Data Conf.*, vol. 13, 2016, p. 13.
[9] A. A. Omar, M. Z. A. Bhuiyan, A. Basu, S. Kiyomoto, and M. S. Rahman, "Privacy-friendly platform for healthcare data in cloud based on blockchain environment," *Future Gener. Comp. Syst.*, vol. 95, pp. 511–521, 2019.
[10] M. S. Rahman, A. A. Omar, M. Z. A. Bhuiyan, A. Basu, S. Kiyomoto, and G. Wang, "Accountable cross-border data sharing using blockchain under relaxed trust assumption," *IEEE Trans. Eng. Manage.*, pp. 1–15, 2020.
[11] L. Li *et al.*, "Creditcoin: A privacy-preserving blockchain-based incentive announcement network for communications of smart vehicles," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 7, pp. 2204–2220, Jul. 2018.
[12] B. Scott, "How can cryptocurrency and blockchain technology play a role in building social and solidarity finance?," Tech. Rep., UNRISD Working Paper, pp. 1–25, 2016.
[13] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges," *IJ Netw. Secur.*, vol. 19, no. 5, pp. 653–659, 2017.

[14] M. del Castillo, "The dao attacked: Code issue leads to $60 million ether theft," 2016. [Online]. Available: http://www. coindesk. com/dao-attacked-code-issue-leads-60-million-ether-theft
[15] H. Nabilou, "How to regulate bitcoin? decentralized regulation for a decentralized cryptocurrency," *Int. J. Law Inf. Technol.*, vol. 27, no. 3, pp. 266–291, 2019.
[16] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proc. 6th Int. Conf. Principles Secur. Trust*, 2017, pp. 164–186.
[17] "A Lottery Contract Example," 2019. [Online]. Available: https://ethereum.stackexchange.com/questions/52159/lottery-contract-for-an-erc20-token
[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, ACM, 2016, pp. 254–269.
[19] "Mythril project," 2019. [Online]. Available: https://github.com/ConsenSys/mythril
[20] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[21] "Ethereum virtual machine operation codes," 2019. [Online]. Available: https://ethervm.io
[22] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF Con*, vol. 25, p. 11, 2017.
[23] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.
[24] K. Bhargavan *et al.*, "Formal verification of smart contracts: Short paper," in *Proc. ACM Workshop Program. Lang. Anal. Secur.*, 2016, pp. 91–96.
[25] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proc. 22nd ACM sigkdd Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 785–794.
[26] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, 1997.
[27] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[28] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Process. Lett.*, vol. 9, no. 3, pp. 293–300, 1999.
[29] T. M. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967.
[30] "Ethereum official website," 2020. [Online]. Available: https://etherscan.io
[31] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2017, pp. 520–535.
[32] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Proc. Int. Conf. Principles Secur. Trust*, 2018, pp. 243–269.
[33] E. Hildenbrandt *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *Proc. IEEE 31st Comput. Secur. Found. Symp.*, 2018, pp. 204–217.
[34] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018, pp. 1–12.
[35] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 259–269.
[36] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.
[37] J. Song, H. He, Z. Lv, C. Su, G. Xu, and W. Wang, "An efficient vulnerability detection model for ethereum smart contracts," in *Proc. Netw. Syst. Secur. - 13th Int. Conf.*, Sapporo, Japan, 2019, pp. 433–442.
[38] W. Wang *et al.*, "Constructing features for detecting android malicious applications: Issues, taxonomy and directions," *IEEE Access*, vol. 7, pp. 67602–67631, 2019.
[39] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Trans. Inf. Forensics Secur.*, vol. 9, no. 11, pp. 1869–1882, Nov. 2014.
[40] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Gener. Comput. Syst.*, vol. 78, pp. 987–994, 2018.
[41] W. Wang, M. Zhao, and J. Wang, "Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network," *J. Ambient Intell. Humanized Comput.*, vol. 10, no. 8, pp. 3035–3043, 2019.
[42] X. Wang, W. Wang, Y. He, J. Liu, Z. Han, and X. Zhang, "Characterizing android apps' behavior for effective detection of malapps at large scale," *Future Gener. Comp. Syst.*, vol. 75, pp. 30–45, 2017.
[43] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang, "Privacy risk analysis and mitigation of analytics libraries in the android ecosystem," *IEEE Trans. Mobile Comput.*, vol. 19, no. 5, pp. 1184–1199, May 2020.
[44] W. Wang, X. Guan, X. Zhang, and L. Yang, "Profiling program behavior for anomaly intrusion detection based on the transition and frequency property of computer audit data," *Comput. Secur.*, vol. 25, no. 7, pp. 539–550, 2006.
[45] X. Liu, J. Liu, W. Wang, Y. He, and X. Zhang, "Discovering and understanding android sensor usage behaviors with data flow analysis," *World Wide Web*, vol. 21, no. 1, pp. 105–126, 2018.
[46] W. Wang, Y. Shang, Y. He, Y. Li, and J. Liu, "Botmark: Automated botnet detection with hybrid analysis of flow-based and graph-based traffic behaviors," *Inf. Sci.*, vol. 511, pp. 284–296, 2020.
[47] W. Wang, X. Guan, and X. Zhang, "Processing of massive audit data streams for real-time anomaly intrusion detection," *Comput. Commun.*, vol. 31, no. 1, pp. 58–72, 2008.
[48] W. Wang and R. Battiti, "Identifying intrusions in computer networks with principal component analysis," in *Proc. 1st Int. Conf. Availability, Rel. Secur.*, Austria, 2006, pp. 270–279.
[49] W. Wang, X. Guan, and X. Zhang, "A novel intrusion detection method based on principle component analysis in computer security," in *Proc. Adv. Neural Netw. Int. Symp. Neural Netw., Part II*, Dalian, China, 2004, pp. 657–662.
[50] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[51] L. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *Proc. 41st Int. Conv. Inf. Commun. Technol., Electron. Microelectronics*, 2018, pp. 1545–1550.
[52] J. Bi and C. Zhang, "An empirical comparison on state-of-the-art multiclass imbalance learning algorithms and a new diversified ensemble learning scheme," *Knowl.-Based Syst.*, vol. 158, pp. 81–93, 2018.
[53] "Oyente project." 2019. [Online]. Available: https://github.com/melon-project/oyente
[54] W. B. Cavnar, and J. M. Trenkle, "N-gram-based text categorization," in *Proc. SDAIR-94, 3rd Annu. Symp. Document Anal. Inf. Retrieval*, 1994. [Online]. Available: https://www.cis.uni-muenchen.de/~stef/seminare/sprachenidentifizierung/cavnar_trenkle.pdf
[55] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD Explorations Newslett.*, vol. 6, no. 1, pp. 20–29, 2004.
[56] J. Friedman, T. Hastie, R. Tibshirani *et al.*, "Additive logistic regression: A statistical view of boosting (with discussion and a rejoinder by the authors)," *Ann. Statist.*, vol. 28, no. 2, pp. 337–407, 2000.



Wei Wang received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, Xi'an, China, in 2006. He is a Full Professor with the Department of Information Security, Beijing Jiaotong University, Beijing, China. He is currently also affiliated with the Division of Computer, Electrical and Mathematical Sciences and Engineering (CEMSE), King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia. He was a Postdoctoral Researcher with the University of Trento, Trento, Italy, from 2005 to 2006. He was a Postdoctoral Researcher with TELECOM Bretagne and INRIA, France, from 2007 to 2008. His main research interests include mobile, computer, and network security. He was a European ERCIM Fellow with the Norwegian University of Science and Technology (NTNU), Norway, and the Interdisciplinary Centre for Security, Reliability, and Trust (SnT), University of Luxembourg, from 2009 to 2011. He has authored or coauthored more than 80 peer-reviewed papers in various journals and international conferences. He is an Editorial Board member of *Computers & Security* and a Young AE of *Frontiers of Computer Science*.

Jingjing Song received the B.S. degree from Yanshan University, Qinhuangdao, China, in 2017. She is currently working toward the M.S. degree with the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing, China. Her main research interests include blockchain and Ethereum security.

Guangquan Xu (Member, IEEE) received the Ph.D. degree from Tianjin University, Tianjin, China, in March 2008. He is a Full Professor with the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University. His research interests include cybersecurity and trust management. He is a member of the CCF.

Yidong Li received the B.Eng. degree in electrical and electronic engineering from Beijing Jiaotong University, Beijing, China, in 2003, and the M.Sc. and Ph.D. degrees in computer science from the University of Adelaide, Adelaide, SA, Australia, in 2006 and 2010, respectively. He is the Vice-Dean and a Professor with the School of Computer and Information Technology, Beijing Jiaotong University. He has authored and coauthored more than 80 research papers in various journals (such as the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS), and refereed conferences. He has also coauthored/coedited five books (including proceedings) and contributed several book chapters. His research interests include big data analysis, privacy preserving and information security, data mining, social computing, and intelligent transportation. He has organized several international conferences and workshops and has also served as a program committee member for several major international conferences such as PAKDD, NFOSCALE, WAC, SAC, PDCAT, DANTH, and PAAP.

Hao Wang received the Ph.D. degree in computer science and engineering from the South China University of Technology, Guangzhou, China, in 2006. He is currently an Associate Professor with the Norwegian University of Science and Technology, Norway. He has authored or coauthored 130+ papers in reputable international journals and conferences. His current research interests include big data analytics, industrial Internet of Things, high performance computing, safety-critical systems, and communication security.

Dr. Wang is a member of the IEEE IES Technical Committee on Industrial Informatics. He served as a TPC Co-Chair for the IEEE DataCom 2015, IEEE CIT 2017, and ES 2017. He served as a Reviewer for journals such as the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, the IEEE TRANSACTIONS ON BIG DATA, the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, the IEEE INTERNET OF THINGS JOURNAL, and *ACM Transactions on Multimedia Computing, Communications, and Applications*.

Chunhua Su received the B.S. degree from Beijing Electronic and Science Institute, Beijing, China, in 2003, and the M.S. and Ph.D. degrees in computer science from the Faculty of Engineering, Kyushu University, Fukuoka, Japan, in 2006 and 2009, respectively. He is currently a Senior Associate Professor of computer science, University of Aizu, Japan. His research areas include algorithm, cryptography, data mining, and RFID security & privacy.