Blockchain: Research and Applications 5 (2024) 100209

Contents lists available at ScienceDirect

Blockchain: Research and Applications

journal homepage: www.journals.elsevier.com/blockchain-research-and-applications

Research Article

An interpretable model for large-scale smart contract vulnerability detection

Xia Feng [a,*], Haiyang Liu [b], Liangmin Wang [c], Huijuan Zhu [b], Victor S. Sheng [d,*]

[a] City University of Macau, Macau 999078, China
[b] Jiangsu University, Zhenjiang 212013, China
[c] Southeast University, Nanjing 211189, China
[d] Texas Tech University, Lubbock 79409, USA

ARTICLE INFO

Keywords:
Blockchain
Vulnerability detection
Smart contract

ABSTRACT

Smart contracts hold billions of dollars in digital currency, and their security vulnerabilities have drawn a lot of attention in recent years. Traditional methods for detecting smart contract vulnerabilities rely primarily on symbol execution, which makes them time-consuming with high false positive rates. Recently, deep learning approaches have alleviated these issues but still face several major limitations, such as lack of interpretability and susceptibility to evasion techniques. In this paper, we propose a feature selection method for uplifting modeling. The fundamental concept of this method is a feature selection algorithm, utilizing interpretation outcomes to select critical features, thereby reducing the scales of features. The learning process could be accelerated significantly because of the reduction of the feature size. The experiment shows that our proposed model performs well in six types of vulnerability detection. The accuracy of each type is higher than 93% and the average detection time of each smart contract is less than 1 ms. Notably, through our proposed feature selection algorithm, the training time of each type of vulnerability is reduced by nearly 80% compared with that of its original.

1. Introduction

Blockchain, as an emerging technology, has attracted extensive attention all over the world. Benefiting from its characteristics of decentralization, tamper proof, irreversibility, and traceability, it has been widely used in agriculture [1], construction [2], cargo transportation [3] and other areas. “Smart contract” [4,5], which is a computer program running on the blockchain, has become one of the most successful applications of blockchain technology. At present, millions of smart contracts have been deployed on various blockchain platforms, such as EOS [6], Ethereum [7], and Vntchain [8], with the quantity continuing to increase at a rapid pace.

However, with the development of smart contracts, the security problem also follows. There are many factors that cause the security problem of smart contracts, such as the inherent vulnerabilities of their programming language, the open network environment, and others. In addition, owing to the non-tamperability and irreversibility of blockchain, when a smart contract is attacked by hackers, we cannot defend it or prevent the execution of the contract. At present, there are many security worries about smart contracts. For example, hackers exploited the reentrant vulnerability of the decentralized atomic organization (DAO) contract to steal about 60 million dollars from Ethereum in 2016 [9]. In 2017, the Parity multi-signed wallet vulnerability resulted in the freezing of nearly 300 million dollars in Ethereum [10]. Attackers took advantage of the integer overflow vulnerability of the US chain BEC contract and made the token worthless in 2018 [11]. Vulnerability detection and security prevention of smart contracts have become key issues and major challenges that need to be solved urgently.

To solve the security issues of the smart contract, researchers have proposed some feasible methods for detecting smart contract vulnerabilities, and these mainstream methods mainly can be divided into three categories. The first category [12,13] employs traditional static analysis and dynamic execution techniques to identify vulnerabilities. These works need the exploration of all executable paths in a contract or the analysis of the contract’s dependency graphs. For instance, experts must predefine the patterns and rules of vulnerabilities in advance under aca-

* Corresponding authors.
E-mail addresses: xiafeng@cityu.mo (X. Feng), haiyang@stmail.ujs.edu.cn (H. Liu), liangmin@seu.edu.cn (L. Wang), hjzhu@ujs.edu.cn (H. Zhu), Victor.Sheng@ttu.edu (V.S. Sheng).

https://doi.org/10.1016/j.bcra.2024.100209
Received 1 January 2024; Received in revised form 26 April 2024; Accepted 28 May 2024
Available online 6 June 2024

demic research standards [14]. Another category [15–17] investigates the use of deep learning models to deal with complex contract data and significantly improves the accuracy of detection. However, because of the black-box nature of deep learning models, these approaches often have poor interpretability. The last category [18,19] demonstrates improved accuracy, reduced time requirements, and enhanced interpretability in the context of smart contract detection. They leveraged the interpretation outcomes to provide the confidence score about vulnerability existence.

We are inspired by previous work and propose a new model suitable for large-scale smart contract detection. This model can detect vulnerabilities in an explainable fashion and adjust the training process according to the interpretation results. The key contributions of this work are:

1) We first explore a feature selection method and then develop a model that can be used for large-scale smart contract vulnerability detection. The model can detect vulnerabilities in an explainable fashion and adjust the training process according to the interpretation results. The inclusion of the feature selection method has resulted in a reduction of approximately 80% in training time, making our model suitable for large-scale smart contract vulnerability detection.
2) To the best of our knowledge, we are the first to leverage the interpretation results to build an adaptive vulnerability detection scheme for smart contracts. The interpretation outcomes are used to select important features, thereby reducing the scale of features for training. With the feature selection strategy, we further use the grid search algorithm to adjust the hyper-parameters. Finally, by identifying and decompiling the most critical features, the model could precisely pinpoint each vulnerability present in the smart contract.
3) We developed the prototype system that works on smart contract vulnerability detection. Our experiments show that the system is time-saving and accurate. The average detection time of a single vulnerability is less than 1 ms. Extensive experiments are performed on all the 40,000 contracts on the Ethereum smart contracts dataset, demonstrating excellent performance with accuracies 93%, 93%, 98%, 100%, 99%, and 100% on six common types of vulnerabilities, respectively. The code can be found at https://github.com/jklujklu/smartcontract.

2. Related works

Considering the huge impact of smart contract vulnerabilities, many research works focus on smart contract vulnerability detection. These works can be roughly categorized into two classes. The first class, including Contractfuzzer [20], Maian [21], and Smartcheck [22], usually uses symbolic execution and classical static analysis to identify vulnerabilities. The main idea of symbolic execution is to replace arbitrary uncertain variables in source code, such as environmental variables and formal parameters, with symbolic values in the process of analysis. Symbolic execution is a powerful generic method for detecting vulnerabilities. However, it may not cover all execution paths, resulting in false negatives. These methods rely on a few fixed expert rules and pose a risk of errors due to manually established patterns. Additionally, the fixed expert rules make it hard to cover certain complex vulnerabilities. Meanwhile, targeted attackers may easily bypass the fixed patterns using several tricks.

The second class of works, including DeeSCVHunter [23], Peculiar [24], and CBGRU [25], explores deep learning models to deal with smart contracts. For instance, Yu et al. [23] developed a systematic and modular framework for smart contract vulnerability detection called DeeSCVHunter. They proposed the concept of vulnerability candidate slicing that can significantly improve the performance of deep learning models. Wu et al. [24] proposed a new tool named Peculiar, which uses the technique of data flow graphs to improve detection. Zhang et al. [25] proposed a hybrid learning model called CBGRU, which combines the advantages of different models and improves the accuracy of vulnerability detection. Due to the black-box nature of deep learning, these approaches fail to encode useful expert knowledge, and most of them have poor explainability.

Recently, there are some studies that add interpretation models after training, making the model interpretable. For instance, Liu et al. [18] proposed a method called AME, which combined expert rules with neural networks to improve detection accuracy. They also used interpretable weight to explain the importance of different features. However, the solution with neural networks needs a significant amount of smart contracts to learn the weight of features. As a result, it is time-consuming and unable to promptly identify the vulnerability of a newly deployed smart contract. To solve these issues, we make use of the interpretation results. By seeking an explainable solution that could calculate the weights of different features, we select the critical features of the datasets. The significant increase in the learning speed could be achieved by the feature selection algorithm, as does the improvement of accuracy.

3. Vulnerabilities

We intend to build an interpretable model that can discover vulnerabilities in large-scale smart contracts. In this model, we concentrate on six types of vulnerabilities shown as follows.

3.1. Integer overflow and integer underflow vulnerabilities

Generally, the definition of integer adopts different formats, such as unsigned integers and signed integers, and each format of integer has an upper limit and a lower limit. For an 8-bit unsigned integer, the minimum value is 0 and the maximum value is 255. During the running of the program, if the integer exceeds this range, there is an integer overflow or overflow vulnerability.

When an integer overflow or underflow happens, there is no prespecified action that is set to take place, which is most likely to result in serious errors. The contract BeautyChain [11] is an example of leveraging an integer overflow as a vulnerability to launch an attack.

3.2. Transaction-ordering dependency (TOD) vulnerability

A block in the blockchain includes a set of transactions that it executes, and hence the block state is updated several times in each epoch. Only the miner who mines the block can decide the order of these transactions, consequently the order of updates. As a result, the final state of a contract depends on how the miner orders the transactions invoking it. Such contracts are called transaction-ordering dependent contracts.

A noteworthy example of a contract with a TOD vulnerability is Bancor [26]. Miners would be able to front-run any transactions on Bancor as they are permitted to re-order transactions within a block they have mined.

3.3. Callstack depth attack vulnerability

In Solidity, the permitted maximum callstack depth is 1023. It means any function can fail at any time if its call exceeds the maximum range. Attackers can trigger a recursive function by constructing specific input data or attack code that repeatedly calls a recursive function until the stack memory reaches its limit, causing the program to crash or execute malicious code, such as transfer function $send()$.

3.4. Timestamp dependency

When a function utilizes a block timestamp as a condition to perform important activities, such as generating random numbers using

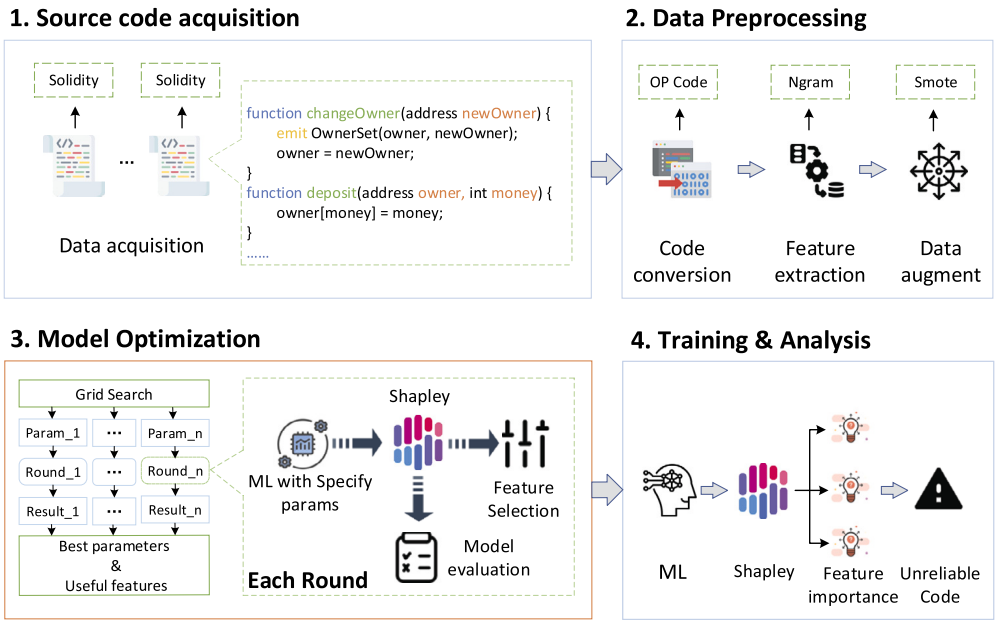

Fig. 1. Framework of our proposed model.

Table 1
Summary of symbols and notations.

Symbol	Description
p_n	Different combinations of all hyper-parameters
x_m	Partial features
θ	Threshold of Shapley values
TP	The number of positive samples correctly predicted
FP	The number of negative samples incorrectly predicted
$Score\_i$	Precision

the block timestamp of a future block as the source to determine the winner of a game, block timestamp dependency occurs. The miner that mines the block has the freedom to set the timestamp of the block as long as it is within a short time interval. As a result, miners may modify the block timestamp and acquire illicit advantages.

3.5. Reentrancy vulnerability

Smart contracts have the ability to call and use code from other contracts. Submitting an external call is required when activating an external contract or sending cryptocurrency to an account. An attacker might use the external call to coerce the contracts to run reentrant routines, such as calling themselves. As a result, similar to indirect recursive function calls in programming, identical code is executed repeatedly. For example, the DAO attack [9] is the most famous case of a reentrancy hack, which has resulted in the loss of 60 million dollars.

4. The proposed smart contract vulnerability detection framework

To detect smart contract vulnerabilities, we propose a smart contract vulnerability detection framework, as shown in Fig. 1. Briefly, it has only four basic steps: 1) source code acquisition; 2) data preprocessing; 3) model optimization; and 4). final training. Table 1 lists the symbols and the corresponding descriptions. The first two steps encompass the construction of the dataset, serving as a preparation for the subsequent steps. Step 3 involves the optimization process for the model, encompassing two distinct tasks. The first task involves feature selection, wherein an interpretable method is utilized to identify the input features that significantly impact the model's prediction. The second task pertains to hyper-parameter tuning, which aims to compare the evaluation scores of pre-trained models with varying hyper-parameters in order to identify the optimal configuration. In the final step, the optimal parameters and the selected features obtained from the preceding step are utilized for training the model, resulting in the attainment of the final model. Here, we will introduce several key parts in the following paragraphs.

4.1. Data preprocessing

Considering the differences of programmer coding styles and programming languages, we use opcode to construct our dataset. In addition, as is known to all that Ethereum has a virtual machine called Ethereum virtual machine (EVM) which ensures the operation and deployment of smart contracts. EVM can not interpret high-level languages, such as Solidity. As an alternative, it uses and defines more than 140 opcodes to perform different functions. The execution process of a smart contract is shown in Fig. 2.

Specifically, we first convert the source code into bytecode through a compiling process. After compiling, the preliminary dataset can be constructed. However, the dataset lacks some necessary features, such as timing characteristics. Therefore, the n-gram algorithm is used to build a complete dataset. n-gram refers to n words that appear continuously in text. It is a probabilistic language model in view of the 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. With the use of a sliding window, the opcodes are segmented into massive n-gram. Note that it is natural for the dataset to be imbalanced because smart contracts with vulnerabilities are not common. Most smart contracts are safe and have no vulnerabilities. To avoid overfitting in training, synthetic minority over-sampling technique (SMOTE) is used to solve the problem of imbalanced data.

As shown in Fig. 3, the imbalance of labels is particularly obvious. Unlike underflow, reentrancy and other vulnerabilities are not very common, which leads to a small number of examples.

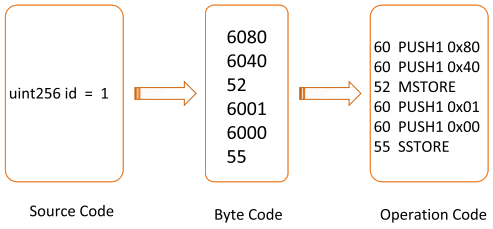

Fig. 2. Execution process of smart contract.

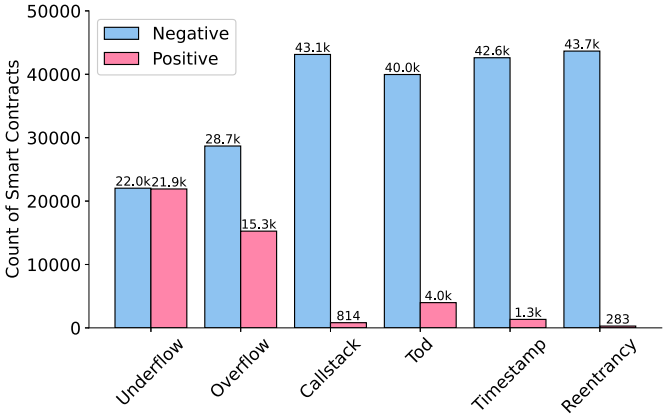

Fig. 3. Overview of raw dataset.

4.2. Model optimization

The model optimization (step 3) is performed from two perspectives: feature selection and hyper-parameter optimization.

4.2.1. Feature selection

To enhance the comprehension of our model, we introduce SHAP (SHapley Additive exPlanations) [27], an interpretation method. It is an additive interpretation method built by Lundberg in 2017, inspired by cooperative game theory. Its core is to calculate the Shapley values and reflect the contribution of each feature to the prediction of the model. We use SHAP to calculate the sum of the Shapley values of each input feature and take it as the result of interpretation. The equation is described as follows:

$$\hat{y} = f_0 + \sum_{i=1}^{M} f_i \tag{1}$$

where $\hat{y}$ is the prediction, f_i is the sum of Shapley values of each feature and f_0 is the mean prediction value of all training samples. For the XGBoost [28] algorithm utilized in our model, the SHAP method is employed to convert the model's prediction values into Shapley values, according to the following equation:

$$\ln \frac{\hat{y}}{1-\hat{y}} = f_0 + \sum_{i=1}^{M} f_i \tag{2}$$

The feature selection algorithm is presented in Algorithm 1. The Shapley value is used as a measure to consider the different contributions of each feature to the model. If the Shapley value is equal to 0, it indicates that the feature does not have any positive or negative impact on the model's output. As a result, the feature is considered insignificant and will be excluded from further analysis. Therefore, we establish the threshold at 0. By establishing a threshold and comparing the Shapley values, the eligible features can be identified and selected.

Algorithm 1: Feature selection algorithm.

Require:
x_n, ▷ All features
θ, ▷ Threshold of Shapley values
Ensure:
X_m, ▷ Features which meet the requirements
Model $f \leftarrow$ Training with x_n
for $i = 1$ to n **do**
  $\hat{y}_i \leftarrow$ equation (1)
  if $|\hat{y}_i| > \theta$ **then**
    $X \leftarrow x_i$
  else
    continue
  end if
end for
return X_m

4.2.2. Hyper-parameter optimization

With the interpretation results, a grid search strategy is used to select the best hyper-parameters automatically. The detail of the approach is shown in Algorithm 2. Specifically, our approach involves defining multiple sets of hyper-parameters, including the learning rate and depth of XGBoost trees. Each hyper-parameter is assigned a distinct range of potential values. These hyper-parameters are then combined to create a grid, where each cell represents a specific combination of hyper-parameters. Through an iterative process, we train and evaluate the model using each option from the grid. The objective is to identify the optimal hyper-parameter configuration that maximizes the model's performance. The hyper-parameters of the model with the highest evaluation score are regarded as the optimal ones. Considering that training with different hyper-parameters may be very time-consuming, we choose to randomly sample the dataset and select only n percent of them to participate in the training of the sub-models.

Algorithm 2: Hyper-parameter optimization algorithm.

Require:
p_n, ▷ Different combinations of all hyper-parameters
x_m, ▷ Partial features
θ, ▷ Threshold of Shapley values
Ensure:
P, ▷ Best combination of hyper-parameters
$Score \leftarrow 0$
for $i = 1$ to n **do**
  Model $f_i \leftarrow$ Training with p_i, x_m
  $\sum_{i=1}^{m} |\hat{y}_i| \leftarrow$ equation (1)
  $Score_i \leftarrow \frac{TP}{TP+FP}$
  if $Score_i > Score$ **and** $\sum_{i=1}^{m} |\hat{y}_i| > \theta$ **then**
    $Score \leftarrow Score_i$
    $P \leftarrow p_i$
  else
    continue
  end if
end for
return P

5. Experiments

In our experiments, we trained models on the training set and verified the performance of the model on the test set. Additional comparative experiments were carried out to demonstrate the superior performance of the proposed model. All experiments were conducted on a computer equipped with an Intel Core i5 CPU at 3.7 GHz, a GPU at 3060, and 16 GB Memory. Machine learning was implemented

Table 2
Some filtered opcode operators.

Function	Operator
Arithmetic operation	ADD, MUL,SUB, DIV SDIV, MOD, ADDMOD MULMOD, EXP, SIGNEXTEND
Condition operation	LT, GT, SLT, SGT EQ, ISZERO
Bit operation	AND, OR, XOR, NOT BYTE, SHL, SHR, SAR
Memory operation	POP, MLOAD, MSTORE SLOAD, JUMPDEST, SSTORE JUMP, PC, MSIZE
Blockchain related	STOP, ADDRESS, BALANCE CREATE, CALL, RETURN BLOCKHASH, SELFBALANCE SELFDESTRUCT, ...

with scikit-learn [29], while other algorithms were implemented with Python.

5.1. Data preparation

From the official website of Ethereum, we collected 43937 verified smart contracts implemented with Solidity [30]. The source code is collected from the Ethereum network.

In our experiment, we labeled the dataset with six classes according to the type of vulnerabilities. For better discrimination, we indexed them in Arabic numerals. For example, 0 represents underflow, 2 represents callstack, 3 represents TOD, 4 represents timestamp, and 5 represents reentrancy. Each vulnerability is independent, and each smart contract may have several vulnerabilities. Therefore, we used a six-digit zero-one vector to label each smart contract. For instance, a contract with the multi-label vector [101000] demonstrates that it has the first and the third types of vulnerabilities. We used Oyente [31] to detect a vulnerability and obtain the multi-label vector for each smart contract. Given that numerous academic papers have utilized Oyente as a benchmark for comparative analysis,we assume that the labels produced by Oyente are reliable according to scholarly research standards.

5.2. Data preprocessing

1) Opcodes & simplification: We used Remix [32], an online Ethereum integrated development environment (IDE), to compile the contracts and transfer them into opcodes. As mentioned above, the EVM opcode has more than 140 operators. Actually, not all operators are useful, such as *log*. Therefore, we specified a set of filtering rules. The filtered operators are shown in Table 2.

2) Initial feature extraction: We employed the *n*-gram algorithm for feature extraction. Through a sliding window of size 2, the simplified opcodes turn into many fragments. The size can be different, for example, 1, 3, or others. To verify which size is more appropriate for vulnerability detection, a comparative experiment was conducted. The results show that with the sliding window of size 2, the system has the best performance. If 1 is chosen, it only considers the word itself and usually leads to poor performance. As to size 3, owing to the increase in the window size, the number of features has also increased. When n is equal to 2, it is commonly referred to as a bigram; when n is equal to 3, it is referred to as a trigram. If the original string, which necessitates n-gram processing, consists of n words, the generation of bigram feature vectors will result in n^2 vectors, while trigram feature vectors will yield n^3 vectors. It is important to note that while n equaling 3 leads to an increase in the number of features, it also significantly increases the dimensionality, thereby resulting in exponential growth in computational complexity. Therefore, it is not appropriate to use it for large-scale smart contract vulnerability detection.

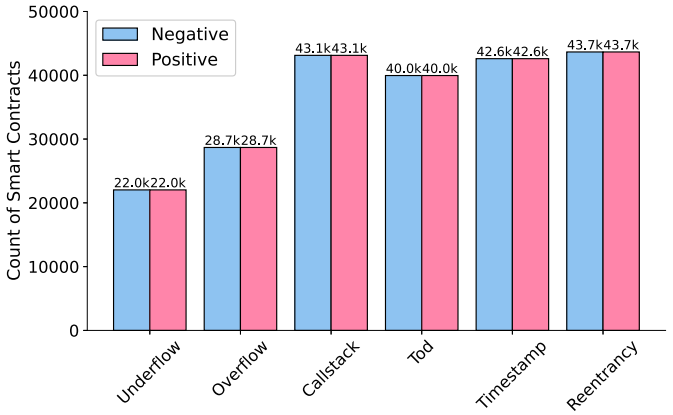

Fig. 4. Overview of the dataset after *n*-gram & synthetic minority over-sampling technique (SMOTE).

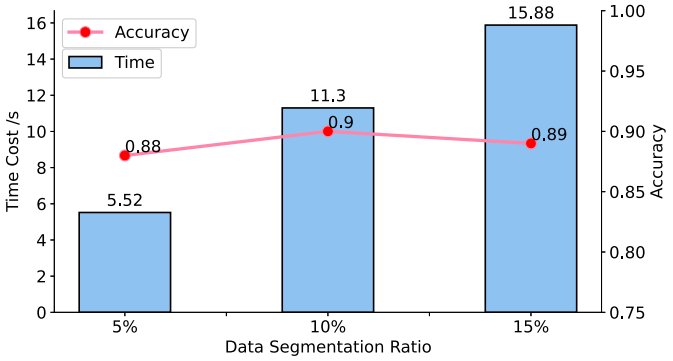

Fig. 5. Comparison of accuracy and time cost under different pretrained parameters.

3) Data augmentation: In this work, as we mentioned before, the training sets are imbalanced because classification categories are not equally represented. To improve the model performance, SMOTE [33] is used in this work. SMOTE is an oversampling technique, interpolating between the minority examples to generate extra ones for each type of vulnerabilities. The overview of the processed dataset is shown in Fig. 4.

5.3. Training process

First, we divide the dataset into training sets and evaluating sets according to the scale of 7:3. Then, n ($n \in \{5, 10, 15\}$) percent of the training sets is selected to train the sub-models. The reason for selecting a 5% interval is based on the practical observation that the *n*-gram algorithm used in our experiment does not generate a large feature dimension. In fact, only a few items from the generated features have a significant impact on the model's output for each category. Therefore, choosing a smaller dataset size is sufficient to cover the majority of effective features. Moreover, the smaller dataset size allows for the faster model training. In this process, the XGBoost classifier with default parameters is regarded as the backbone model. The results of different segmentation ratios are shown in Fig. 5. We can conclude that training with 10% of the training data can achieve the balance between the training time cost and accuracy. Although the results of these three options are not much different, considering that the dataset may be larger in the future, we finally chose 10% as a basic parameter of our system.

Second, we try to find the best hyper-parameters for our model. We choose three hyper-parameters of XGBoost for optimization, which are learningRate, numLeaves, and maxDepth. Their value ranges are shown in Eq. (3).

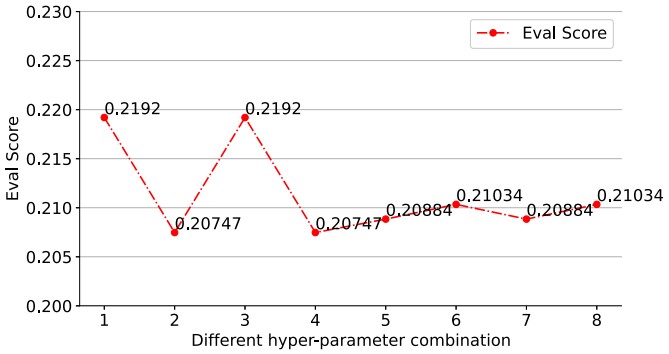

Fig. 6. Comparison of evaluation score under different hyper-parameters.

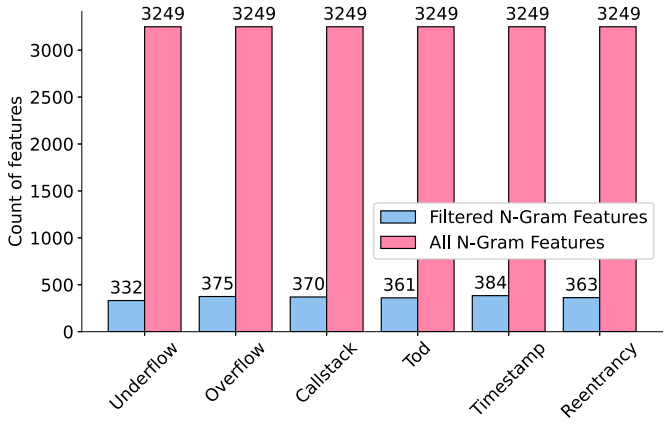

Fig. 7. Overview of features after *n*-gram.

$$\begin{aligned} &learning\,Rate\ \epsilon\ \{0.2, 0.1\} \\ &numLeaves\ \epsilon\ \{25, 35\} \\ &maxDepth\ \epsilon\ \{10, 12\} \end{aligned} \tag{3}$$

Our model iterates through each option and employs the same dataset for training purposes. Each iteration will produce a sub-model. Then, we calculate the sum of Shapley values and use a threshold to judge the usability of the sub-model. If passed, the precision score is regarded as the evaluation score of the sub-model. Take underflow vulnerability as an example, Fig. 6 shows the evaluation scores of different hyper-parameters. The x-axis represents the combination of hyper-parameters used in training, which starts from 1 (which means $\{0.2, 25, 10\}$) to 8 (which means $\{0.1, 35, 12\}$).

The most suitable model hyper-parameters can be obtained from Fig. 6. For underflow vulnerability detection, the hyper-parameter combination of $\{0.2, 25, 12\}$ can be selected as the optimal hyper-parameters. By utilizing the optimal hyper-parameters, we train the model and achieve an accuracy of 92% on the partial dataset.

The Shapley analysis output also provides insights into the significance of each n-gram feature. A Shapley value of zero indicates that the corresponding feature has no impact on the model prediction. Consequently, we can consider removing such features from the feature space as they are deemed redundant. The number of selected features is shown in Fig. 7.

Also, we conducted extra training with all the n-gram features compared with our scheme. The results are shown in Fig. 8. We can see that using the filtered features for training has no significant impact on the accuracy of the model.

Fig. 9 shows the time required for all processes. The right pink bar represents the time required for training with all features while the left bar represents the time taken for training with filtered features. The left bar is divided into three parts: 1) The light-green section represents the time taken for pretraining; 2) the green section represents the time spent on explaining with SHAP, where the results of SHAP are used to screen and select features; 3) the blue section represents the time taken for the final training. Indeed, Fig. 9 shows that our proposed feature extraction scheme significantly reduces the time required for all processes. This reduction in time can lead to improved efficiency and faster model training while still maintaining high accuracy.

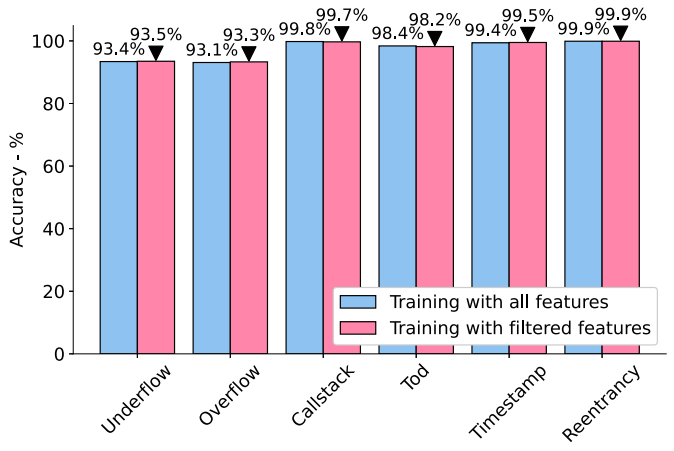

Fig. 8. Accuracy comparison between training with all features and partial features.

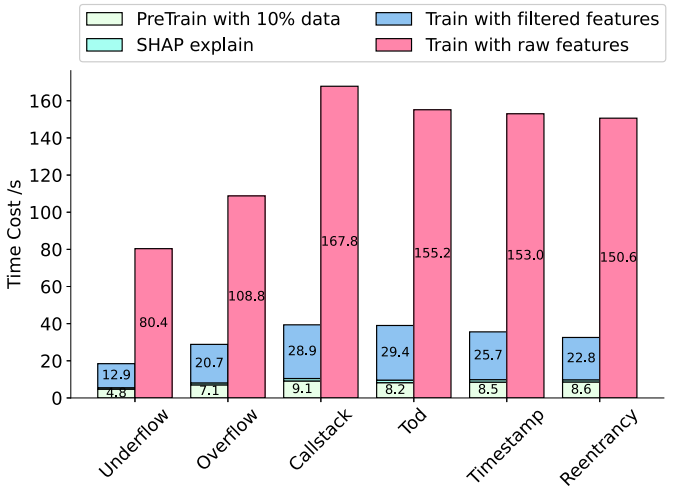

Fig. 9. Time comparison between training with all features and partial features.

5.4. Evaluation index

To evaluate the effectiveness of the proposed method, we choose Precision, Recall, and F1-Score as evaluation indices of classification. Their equations are as follows:

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP}+\text{FP}}, \\ \text{Recall} &= \frac{\text{TP}}{\text{TP}+\text{FN}}, \\ \text{F1-Score} &= \frac{2\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}. \end{aligned} \tag{4}$$

The evaluation results are shown in Table 3, where the value 0 represents a normal smart contract and the value 1 represents a vulnerable smart contract. The results indicate that our proposed scheme achieves an accuracy of over 93% for six types of vulnerabilities. Additionally, for certain uncommon vulnerabilities, the accuracy reaches an impres-

Table 3
Evaluation results.

Vulnerability	Underflow		Overflow		TOD		Callstack		Timestamp		Reentrancy	
	0	1	0	1	0	1	0	1	0	1	0	1
Precision	0.93	0.93	0.92	0.95	0.97	0.99	1.00	1.00	0.99	1.00	1.00	1.00
Recall	0.93	0.93	0.95	0.92	0.99	0.97	1.00	1.00	1.00	0.99	1.00	1.00
F1-Score	0.93	0.93	0.93	0.93	0.98	0.98	1.00	1.00	0.99	0.99	1.00	1.00
Accuracy		**0.93**		**0.93**		**0.98**		**1.00**		**0.99**		**1.00**

TOD: transaction-ordering dependency

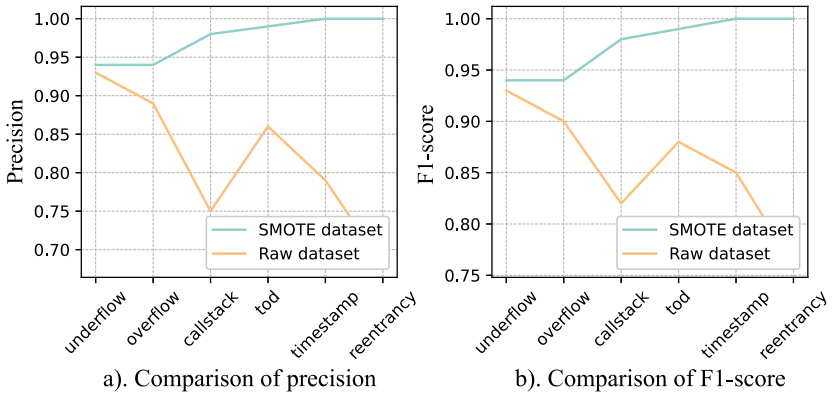

Fig. 10. The comparison results of different datasets.

sive 99%. These high accuracy rates demonstrate the credibility of our system.

Comparing among the six different vulnerabilities, it is observed that the accuracies of underflow and overflow vulnerabilities are notably lower than those of the other vulnerabilities. The reason for this difference is the significant disparity in the number of real datasets available for these vulnerabilities. This quantity gap can be clearly seen in Fig. 3. Despite using SMOTE to address the data imbalance, the limited number of samples remains a crucial factor that impacts the model's performance.

5.5. Performance evaluation

We conducted experiments and trained models on two different datasets: the SMOTE dataset and the raw dataset. We calculated the precision and F1-scores of the models for six types of vulnerabilities. We plotted the experimental results on Fig. 10. Fig. 10a shows the comparison of precision corresponding to the six types of vulnerabilities, while Fig. 10b shows the comparison of F1-scores. It can be observed that when using the raw dataset, both precision and F1-score values are significantly lower compared with the SMOTE dataset. This indicates that the trained model performs poorly and fails to accurately distinguish smart contracts with vulnerabilities. Specifically, the performance of the trained model on the raw dataset decreases by more than 20% in the case of highly imbalanced classes, such as timestamp dependency vulnerability and reentrancy vulnerability.

Based on these results, we continued to utilize SMOTE technique. Additionally, we integrated *k*-fold cross validation to enhance the representativeness of the data. This methodology entails partitioning the dataset into training and validation sets for each subset, training the model, assessing model performance on each validation set, and iteratively repeating this process until all data have been employed for training and validation. By adhering to these procedures, we can enhance the alignment of the generated data with actual smart contracts.

In addition, we compared our system with the schemes proposed in previous studies. Specifically, we conducted a comparative analysis with four open-source schemes based on the same dataset and environment:

1) **DeeSCVHunter**[1] is proposed by Yu et al. [23], can only detect reentrancy and timestamp dependency vulnerabilities.
2) **Peculiar**[2] is proposed by Wu et al. [24], can only detect reentrancy vulnerability.
3) **AME**[3] is proposed by Liu et al. [18], can only detect reentrancy, timestamp dependency, and infinite loop vulnerabilities.
4) **CBGRU**[4] is proposed by Zhang et al. [25], can detect reentrancy, integer overflow, infinite loop, callstack depth, timestamp dependency, and integer underflow vulnerabilities.

Taking into account the fact that previous studies have focused on different target vulnerabilities, we chose to compare the performance of our proposed scheme specifically for the timestamp dependency vulnerability and reentrancy vulnerability. The results of this comparison are illustrated in Figs. 11 and 12.

Fig. 11 focuses on the analysis of reentrancy vulnerabilities, using two metrics: ACC (accuracy) and F1-score for comparison. The results clearly indicate that our proposed approach outperforms the other four methods in both metrics.

Indeed, when there is a significant class imbalance in the dataset, the model tends to predict the majority class more frequently. This can result in high accuracy but poor performance in recognizing minority classes, thus impacting the F1-score. As evident from Fig. 3, the number of smart contracts with reentrancy vulnerabilities is extremely small, only 283, less than one percent of the dataset. Therefore, the other methods may exhibit significant differences between their F1-scores and

[1] https://github.com/MRdoulestar/DeeSCVHunter.
[2] https://github.com/wuhongjun15/Peculiar.
[3] https://github.com/Messi-Q/AMEVulDetector.
[4] https://github.com/xiaoaochen/CBGRU.

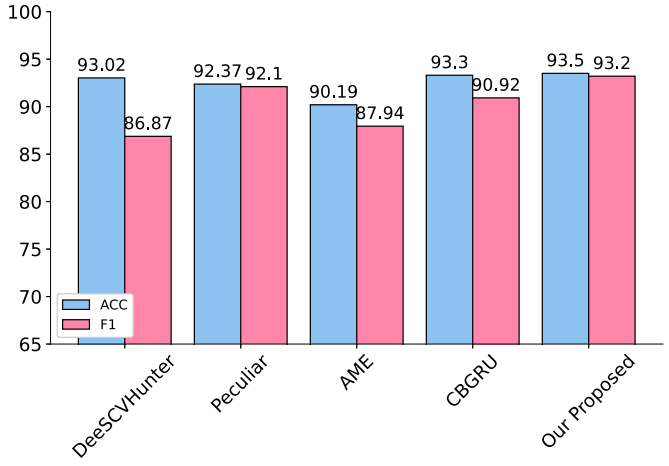

Fig. 11. Comparison of different approaches for reentrancy vulnerability.

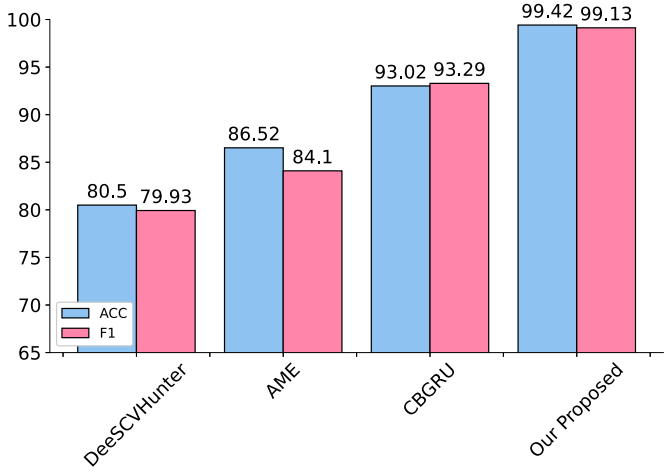

Fig. 12. Comparison of different approaches for timestamp dependency vulnerability.

accuracies due to the limited number of samples available for this specific vulnerability.

However, our approach addresses this issue by employing the SMOTE oversampling technique to balance the dataset. This helps to improve the performance of our approach, particularly in terms of the F1-score, by providing better representation for the minority class.

Fig. 12 demonstrates the superiority of our approach in detecting timestamp dependency vulnerabilities. Unlike reentrancy vulnerabilities, our approach outperforms the other four methods significantly in this case. This is because the model optimization process in Algorithm 1 enables our approach to consistently find the most suitable hyper-parameters for identifying timestamp dependency vulnerabilities, which is a unique advantage that other methods do not possess.

5.6. Vulnerability analysis

In this subsection, we use the interpretation results to analyze each vulnerability. Since different vulnerabilities are interpreted similarly, the timestamp dependency vulnerability is taken as an example below.

Fig. 13 is a forced plot based on the first smart contract with timestamp dependency vulnerability. We can see that all features are grouped to either side of the predicted result. The base value (the mean Shapley value) is drawn in the center of the graph. Each feature is evaluated based on its individual contribution as a force. The blue color indicates a positive impact on increasing the base value towards the final output, while the red color signifies the opposite effect.

Table 4
SHAP importance.

Index	Operation code	Value	Importance(%)
2847	sload timestamp	1.078477	9.326380
367	call pop	0.343625	2.971578
3182	timestamp lt	0.340533	2.944843
3174	timestamp gt	0.242543	2.097449
2418	pop iszero	0.217558	1.881385
1496	jumpdest timestamp	0.160475	1.387746
1355	iszero jumpi	0.152916	1.322377
1465	jumpdest jump	0.123903	1.071484
10	add calldataload	0.123526	1.068222
948	div iszero	0.1109	0.959033

Fig. 14 shows the feature importance for the global model. It is a visual description of interpretation model output, and Table 4 lists the calculated Shapley values of the top 10 features. All of them are sorted according to their Shapley values.

Feature importance is the most popular explanation technique. It shows the influence of all features on the prediction of the model which can give us better interpretability of the data. However, from it, we only look at the absolute values of importance, but cannot know which feature positively or negatively influences the model. Fig. 15 solves this problem.

In Fig. 15, there are three dimensions of data: 1) The left vertical axis denotes feature names, which are ordered based on importance from top to bottom; 2) the horizontal axis represents the magnitude of the Shapley values for predictions; 3) the color represents the feature value (number of occurrences of this feature) in each smart contract. The red color indicates higher feature values, while the blue color means lower values.

By observing the different colors and their corresponding Shapley values, we can assess the impact of each feature on smart contract vulnerability detection. For example, if the red region of a feature corresponds to positive Shapley values, we can conclude that the feature tends to push the model toward producing positive results. Specifically, for the feature sload timestamp, as the feature value increases (the color changes from blue to red), the corresponding Shapley value also increases continuously. This, in turn, makes the model more inclined to output “1” (a vulnerable smart contract). Similar behavior can be observed for features like timestamp lt and timestamp gt. On the other hand, features like iszero jumpi and jumpdest jump exhibit the opposite trend.

6. Conclusion

This study proposes an efficient system for large-scale smart contract vulnerability detection. Our experiment results showed that it is time-saving and reliable. Different from other schemes, our system can make full use of the interpretation result and dynamically adjust the training process according to it. To reduce training time, we developed a feature selection algorithm, and the training time of each vulnerability is reduced by nearly 80% compared with that of the original. To ensure the performance of the model, we develop a hyper-parameter optimization algorithm, and the accuracy of each vulnerability is higher than 93%.

In future work, to further improve the performance of our system, we will explore more effective features to describe the characteristics of smart contracts and use more real smart contracts to evaluate our model.

CRediT authorship contribution statement

Xia Feng: Writing – review & editing, Writing – original draft, Investigation, Conceptualization. **Haiyang Liu:** Writing – original draft, Formal analysis, Data curation. **Liangmin Wang:** Project administration,

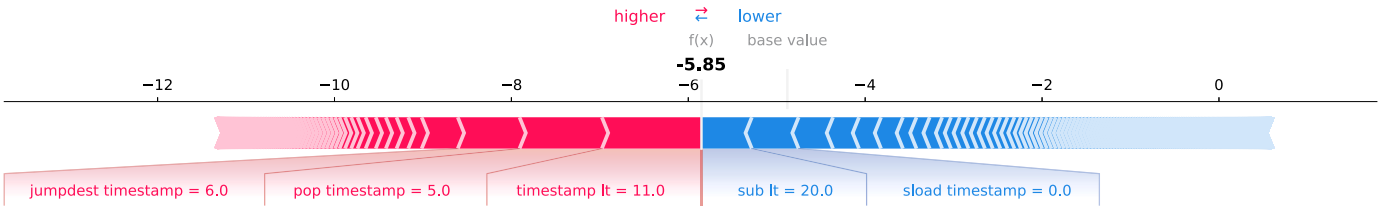

Fig. 13. Force plot on the first smart contract with timestamp dependency vulnerability.

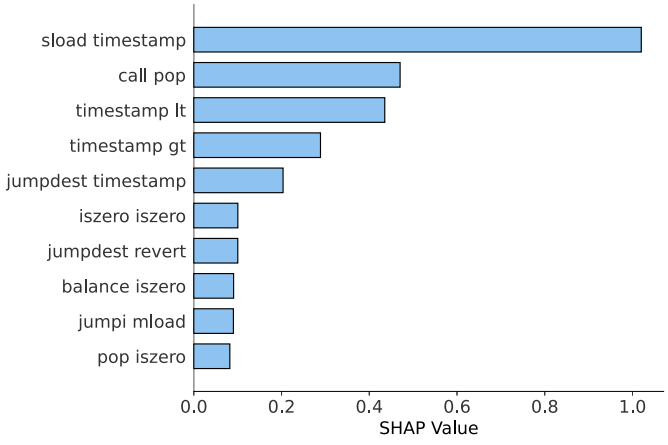

Fig. 14. Global feature importances with SHAP.

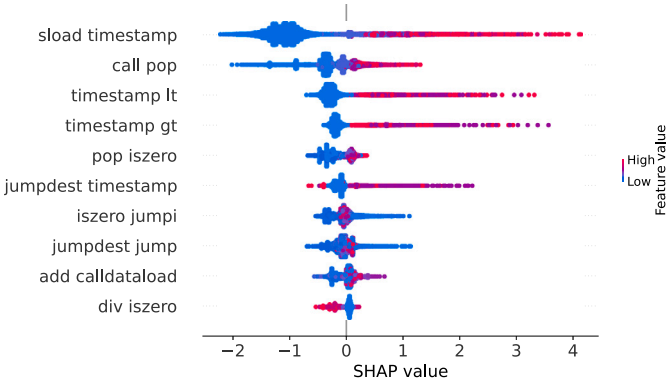

Fig. 15. Summary plot with SHAP.

Methodology, Conceptualization. **Huijuan Zhu:** Visualization, Validation, Software. **Victor S. Sheng:** Writing – review & editing, Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This document is the result of the research project funded by the National Natural Science Foundation of China No. 61902157 and No. 62002139.

References

[1] K. Adel, A. Elhakeem, M. Marzouk, Decentralizing construction AI applications using blockchain technology, Expert Syst. Appl. 194 (2022) 116548, https://doi.org/10.1016/j.eswa.2022.116548.
[2] I. Eluubek kyzy, H. Song, A. Vajdi, et al., Blockchain for consortium: a practical paradigm in agricultural supply chain system, Expert Syst. Appl. 184 (2021) 115425, https://doi.org/10.1016/j.eswa.2021.115425.
[3] M. Baygin, O. Yaman, N. Baygin, et al., A blockchain-based approach to smart cargo transportation using UHF RFID, Expert Syst. Appl. 188 (2022) 116030, https://doi.org/10.1016/j.eswa.2021.116030.
[4] V. Buterin, A next-generation smart contract and decentralized application platform, White Pap. 3 (37) (2014).
[5] L. Ouyang, S. Wang, Y. Yuan, et al., Smart contracts: architecture and research progresses, Acta Autom. Sin. 45 (2019) 445–457, https://doi.org/10.16383/j.aas.c180586.
[6] EOS, Eosio blockchain software, https://eos.io/, 2018. (Accessed 27 June 2022).
[7] Etherscan, Ethereum (eth) blockchain explorer, https://etherscan.io/, 2015. (Accessed 27 June 2022).
[8] VNTChain, Vnt chain official website, https://scan.vntchain.io/, 2019. (Accessed 27 June 2022).
[9] Wikipedia, The dao, https://en.wikipedia.org/wiki/TheDAO, 2022. (Accessed 27 June 2022).
[10] Lorenz, An in-depth look at the parity multisig bug, https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug, 2017. (Accessed 27 June 2022).
[11] Etherscan, Beautychain integer overflow, https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d/, 2018. (Accessed 27 June 2022).
[12] X. Hu, Y. Zhuang, S.W. Lin, et al., A security type verifier for smart contracts, Comput. Secur. 108 (2021) 102343, https://doi.org/10.1016/j.cose.2021.102343.
[13] Z. Alom, B.C. Singh, Z. Aung, et al., Knapsack graph-based privacy checking for smart environments, Comput. Secur. 105 (2021) 102240, https://doi.org/10.1016/j.cose.2021.102240.
[14] Q. Zhou, K. Zheng, K. Zhang, et al., Vulnerability analysis of smart contract for blockchain-based IoT applications: a machine learning approach, IEEE Int. Things J. 9 (2022) 24695–24707, https://doi.org/10.1109/JIOT.2022.3196269.
[15] T. Hu, B. Li, Z. Pan, et al., Detect defects of solidity smart contract based on the knowledge graph, IEEE Trans. Reliab. 73 (1) (2023) 186–202, https://doi.org/10.1109/TR.2023.3233999.
[16] S. Kalra, S. Goel, M. Dhawan, et al., Zeus: analyzing safety of smart contracts, in: NDSS, 2018, pp. 1–12.
[17] L. Zhang, Y. Li, R. Guo, et al., A novel smart contract reentrancy vulnerability detection model based on BiGAS, J. Signal Process. Syst. 96 (3) (2024) 215–237, https://doi.org/10.1007/s11265-023-01859-7.
[18] Z. Liu, P. Qian, X. Wang, et al., Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion, arXiv, 2021, preprint, arXiv:2106.09282.
[19] C. Sendner, H. Chen, H. Fereidooni, et al., Smarter contracts: detecting vulnerabilities in smart contracts with deep transfer learning, in: NDSS, 2023.
[20] B. Jiang, Y. Liu, W.K. Chan, et al., 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, https://doi.org/10.1145/3238147.3238177.
[21] I. Nikolić, A. Kolluri, I. Sergey, et al., Finding the greedy, prodigal, and suicidal contracts at scale, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACM, 2018, pp. 653–663, https://doi.org/10.1145/3274694.3274743.
[22] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, et al., Smartcheck: static analysis of Ethereum smart contracts, in: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, ACM, 2018, pp. 9–16, https://doi.org/10.1145/3194113.3194115.
[23] X. Yu, H. Zhao, B. Hou, et al., Deescvhunter: a deep learning-based framework for smart contract vulnerability detection, in: Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN), IEEE, 2021, pp. 1–8, https://doi.org/10.1109/IJCNN52387.2021.9534324.

[24] H. Wu, Z. Zhang, S. Wang, et al., Peculiar: smart contract vulnerability detection based on crucial data flow graph and pre-training techniques, in: Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2021, pp. 378–389, https://doi.org/10.1109/ISSRE52982.2021.00047.
[25] L. Zhang, W. Chen, W. Wang, et al., Cbgru: a detection method of smart contract vulnerability based on a hybrid model, Sensors 22 (9) (2022) 3577, https://doi.org/10.3390/s22093577.
[26] I. Bogatyy, Implementing Ethereum trading front-runs on the bancor exchange in python, https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798, 2017. (Accessed 27 June 2022).
[27] S.M. Lundberg, A unified approach to interpreting model predictions, arXiv, 2017, preprint, arXiv:1705.07874.
[28] T. Chen, C. Guestrin, Xgboost: a scalable tree boosting system, in: Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 785–794, https://doi.org/10.1145/2939672.2939785.
[29] D. Cournapeau, scikit-learn, https://scikit-learn.org, 2023. (Accessed 27 June 2022).
[30] C. Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, Apress, Berkeley, CA, 2017.
[31] L. Luu, D.-H. Chu, H. Olickel, et al., Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 254–269, https://doi.org/10.1145/2976749.2978309.
[32] Remix, https://remix.ethereum.org/, 2022. (Accessed 27 June 2022).
[33] N.V. Chawla, K.W. Bowyer, L.O. Hall, et al., SMOTE: synthetic minority oversampling technique, J. Artif. Intell. Res. 16 (2002) 321–357, https://doi.org/10.1613/jair.953.