IEEE Open Journal of the Computer Society

Received 3 November 2020; accepted 16 November 2020. Date of publication 24 November 2020; date of current version 14 December 2020. The review of this paper was arranged by Guest Editors Hong-Ning Dai, Huawei Huang, Edith C.H. Ngai, and Salil Kanhere.

Digital Object Identifier 10.1109/OJCS.2020.3039991

Trace-Based Dynamic Gas Estimation of Loops in Smart Contracts

CHUNMIAO LI [1,2], SHIJIE NIE[1], YANG CAO [3], YIJUN YU [4], AND ZHENJIANG HU[5] (Fellow, IEEE)

[1] National Institute of Informatics, Chiyoda-ku 101-8430, Japan
[2] The Graduate University for Advanced Studies(SOKENDAI), Shonan Village, Hayama, Kanagawa 240-0193, Japan
[3] Department of Social Informatics, Kyoto University, Kyoto 6068501, Japan
[4] Department of Computing, The Open University, MK7 8LR Milton Keynes, U.K.
[5] Information Systems Architecture Research Division, National Institute of Informatics, Chiyoda-ku 101-8430, Japan

CORRESPONDING AUTHOR: CHUNMIAO LI (e-mail: chunmiaoli1993@gmail.com)

This work was supported in part by JSPS KAKENHI under Grants JP17H06099, JP18H04093 and 19K20269, in part by EU H2020 Engage KTN, EPSRC EP/T017465/1, EPSRC EP/R013144/1 and Big Code Forensic Analytics in Secure Software Engineering (Royal Society, IES\R1\191138), and in part by CCF-Tencent Open Fund WeBank Special Fund.

ABSTRACT Smart contracts on Ethereum can be used to encode business logic and have been applied to many different areas, such as token exchanges and games. Unlike general programs, the computations of contracts on Ethereum are restricted by the gas limit. If a transaction runs out of the gas limit before an execution finishes, the Ethereum virtual machine throws an out-of-gas exception, and the entire transaction fails, which reverts to the state before the transaction started, although the transaction fee is still deducted. It is therefore, essential to conduct a gas estimation before sending a transaction. Existing studies have mostly failed in estimating the gas for a loop function because the number of iterations of the loops cannot be statically determined. However, we found that a quarter of all contracts have loop functions, and the gas cost for the loops is higher than for the other functions. Therefore, it is necessary to apply a gas estimation for the loop functions. In this study, we propose a gas estimation approach based on the transaction trace to dynamically estimate the gas for the loop functions. Our belief is that we can learn the relationship between the historical transaction traces and their gas costs to estimate the gas for new transactions. We considered three different abstractions of the original transaction trace and fed them to different machine learning models. The results show that our approach is effective in gas estimation and that a random forest can achieve the most accurate estimation.

INDEX TERMS Ethereum, gas estimation, machine learning, out-of-gas, smart contracts.

I. INTRODUCTION

Ethereum [1] is currently the most popular public blockchain, not only because it provides a decentralized, shared ledger, allowing all users to participate in the ledger update activities, but also because it builds a "world computer" that can host and execute programs. These programs are so-called smart contracts [1]. Any user can deploy their contracts to Ethereum by sending a contract creation transaction. Concretely, users construct programs using Solidity, a widely used programming language on Ethereum. These Solidity programs are then compiled to bytecode and stored in the code field of a newly built contract account after the contract creation transaction succeeds. One can send a transaction to the contract account when wanting to call a function on the contract. Once all Ethereum nodes verify the transaction, the Ethereum virtual machine (EVM) will run the contract runtime bytecode on the transaction input data.

An EVM is a stack-based, Turing-complete machine that can program any computation that a Turing Machine can execute, such as a loop. To prevent resource waste from infinite loops and make sure that contract programs can stop at a certain point, the computation effort required to execute the EVM instructions is charged in the gas unit. When a user sends a transaction, the user needs to specify a gas limit attached to the transaction. *GasLimit* is the maximum available amount of gas for a transaction execution. However, if a transaction

execution requires more gas than the gasLimit, the EVM will emit an out-of-gas exception immediately, and all executed operations will be reverted. Meanwhile, the expenses required for purchasing the gas limit are transferred to the beneficiary accounts. The out-of-gas exception accounts for over 90% of all exceptions on Ethereum and causes substantial financial losses [2]. The leading root causes [2] for this exception are users being unfamiliar with the transaction execution mechanism and there being no useful tool for a gas estimation. There are mainly two ways to prevent out-of-gas exceptions. One way is to detect contracts with gas-focused vulnerabilities [3] to prevent users from calling vulnerable functions. The other way is a gas estimation in which, given a transaction, we need to estimate its gas cost. Some studies have been devoted to gas estimations [4]–[7]. For example, Solc[1] statically predicts the gas cost for all contract functions. Marescotti *et al.* [6] apply symbolic model checking methods to detect the worst-case gas cost.

We found that a quarter of all contracts have loop functions, and the gas costs for loops are higher than that for other functions. It is therefore, necessary to conduct a gas estimation for the loop functions. Unfortunately, existing methods mostly fail in estimating the gas cost for transactions to loop functions (i.e., functions containing loops). A static analysis cannot determine the number of iterations of any loops; hence, Gasol [7], a gas estimation tool, fails when the maximal number of iteration times is unbounded. Dynamic methods often send transactions to the local testnet and observe the gas cost, although this gas is not the same as the actual transaction gas cost because the Ethereum mainnet may change and differ from the testnet.

In this paper, we propose a novel approach to achieve a dynamic gas estimation for loop functions based on the transaction execution trace. The belief is that gas costs for new transactions can be predicted based on analyzing the transactions history. Our main idea is to learn the relationship between the transaction trace and gas from the transactions history and apply this to the gas estimation for new transactions. We consider using machine learning algorithms to determine these relations. To the best of our knowledge, we are the *first* to introduce machine learning ideas to a gas estimation. However, it is challenging to implement this idea for two reasons: (1) it can be difficult to know how to collect the traces for a number of specific historical transactions and (2) the traces for executing the loop transactions are extremely long, with the longest trace observed being 382,552. It is difficult to directly feed such a long sequence to any existing learning models.

To address challenge (1), we use an Ethereum-js virtual machine[2] to automatically record a trace when replaying historical transactions in a forked chain. For challenge (2), we take three abstractions of the trace as features and feed them into different learning models. The first abstraction is the frequency for 141 opcodes used on an EVM. The second abstraction is to append a function vector to the end of the first abstraction. Suppose a transaction calls function *f*. The function vector describes the number of occurrences for different structural characteristics in *f*. These two types of abstractions are inputs to three learning models: a random forest, K-nearest neighbor (KNN), and a support vector machine (SVM) for regression (SVR). An EVM charges dynamic gas for 24 opcodes depending on the runtime state. The third abstraction is a dynamic opcode sequence, which is sent to a long short-term memory (LSTM) model. The experimental results show that our approach is effective in a gas estimation for loop functions. The mean absolute percentage error (MAPE) ranges from 0.59 to 67.19 in different learning algorithms. In general, the random forest and KNN can achieve a better prediction accuracy rate than the SVR and LSTM.

Note that the conference version of our paper is provided in [8]. This journal paper extends our previous study in two aspects. First, we collected more loop transaction traces and estimated their gas cost. The estimation results confirmed the observation in our previous paper that a random forest and a KNN can achieve a better estimation accuracy rate than an SVR. Second, our conference paper only proposed two types of features, i.e., the opcode frequency and a dynamic opcode sequence. To improve the estimation accuracy rate, in this study, we combine the opcode frequency and function vector as the third type of feature for a transaction. The function vector describes the number of occurrences of different structural characteristics in the function. Our experimental results show that taking a combination of the opcode frequency and loop function vector as features can lower the prediction error rate compared with two previous types of abstractions.

In summary, the contributions of this paper are as follows.

1) We provide a novel approach to estimating a gas based on the transaction execution trace. The main idea is that the relationship between the transaction trace and gas from historical transactions can be learned to estimate the gas for new transactions. To the best of our knowledge, we are the *first* to use machine learning for a gas estimation.
2) We consider the random forest, KNN, SVR, and LSTM learning models in our experiments. The results show that the random forest and KNN models can achieve a better prediction accuracy rate than the SVR and LSTM.
3) We provide a dataset containing an opcode execution sequence and gas costs for 5718 transactions specially sent to the loop functions. This dataset can be used for later studies on the gas cost of the transactions sent to the loop functions.

The following sections are organized as follows. Section II provides the necessary background for our study. Section III presents the workflow of our trace-based learning method, and Section IV describes the experimental results and limitations. We provide a list of previous related studies in Section V, and give some concluding remarks regarding this research in Section VI.

[1]https://github.com/ethereum/solidity

[2]https://github.com/ethereumjs/ethereumjs-vm

```
1  function getOwnFashions(address _owner) external view returns(uint256[] tokens, uint32[] flags) {
2    require(_owner != address(0));
3    uint256[] storage fsArray = ownerToFashionArray[_owner];
4    uint256 length = fsArray.length;
5    tokens = new uint256[](length);
6    flags = new uint32[](length);
7    for (uint256 i = 0; i < length; ++i) {
8      tokens[i] = fsArray[i];
9      Fashion storage fs = fashionArray[fsArray[i]];
10     flags[i] = uint32(uint32(fs.equipmentId) * 10000 + uint32(fs.quality) * 100 + fs.pos);
11   }
12 }
```

Listing 1: The iteration times of loop is decided runtime.

II. PRELIMINARY

A. GAS MECHANISM ON SMART CONTRACTS

Blockchain is a decentralized shared ledger, and Ethereum [1] is currently the most popular public blockchain. There are two types of accounts on Ethereum: externally owned accounts (i.e., user accounts) and contract accounts. A contract account can store code (i.e., smart contracts) used to encode a business logic. Once a user sends a transaction to a contract account, the contract *opcodes* will be executed on an EVM. Given a transaction, the executional *opcode sequence* in an EVM is called a *transaction trace*. All transactions need to be conducted on all blockchain nodes. To avoid network abuse and some inevitable issues (e.g., infinite loops) caused by the Turing-complete contract language Solidity, all EVM instructions are subject to fees [1]. A fee is measured in gas units.

The gas limit is implicitly deducted from the sender's account balance at a certain gas price before the transaction starts. During the EVM working process, the available gas is reduced by executing the opcodes. Suppose the gas limit is G_1 and the actual execution cost of a transaction is G_2. Note that there is another limit called the block gas limit G_b, which is the maximum amount of gas allowed in a block. In terms of the relationships among G_1, G_2>, and G_b, different transaction execution scenarios are given below:

1) $G_2 < G_b$ and $G_1 >= G_2$: The transaction can be included in a block and is successful. The gas remaining at the end of the transaction is refunded to the sender's account.
2) $G_2 < G_b$ and $G_1 < G_2$: The transaction can be included in a block but fails through an error. The EVM will emit an out-of-gas exception because there is no available gas to support further operations during the transaction execution. At this time, all gas cost is delivered to the miner's account (beneficiary account), and all states applied are reverted right before the transaction starts.
3) $G_2 > G_b$: The transaction cannot be included to a block and fails no matter how large G_1 is.

B. DIFFICULTIES FOR GAS ESTIMATION

In general, there are three types of transactions on Ethereum: (1) money transfer between user accounts, (2) contract deployment, and (3) a function execution on the existing deployed transactions. Note that the gas cost studied in this paper is focused on the third type of transaction, which we call an interactive transaction. The gas cost for an **interactive transaction** (tx) consists of three parts [2]: (a) an intrinsic gas cost, (b) an execution gas cost, and (c) a refund gas cost. The formula used to compute the transaction gas cost is shown in equation 1. The intrinsic gas can be calculated directly according to [1], whereas the execution gas cost is complex and almost impossible to accurately compute.

It is nontrivial to estimate the execution gas cost. First, an EVM charges a gas cost when running a contract program and finishes when it reaches a halted state or runtime exception. Because a halting problem is not decidable [9], there is no way to obtain the exact runtime opcode sequence on an EVM before the transaction execution. Moreover, different EVM versions provide a slightly different gas cost for the same transaction [10]. It is unknown which EVM client the version miner will adopt. In addition, the execution gas may depend on the state of the blockchain [7], which continues updating with new transactions.

$$G(tx) = G_{intrinsic}(tx) + G_{execution}(tx) - G_{refund}(tx) \quad (1)$$

For example, it is difficult to statically estimate an accurate gas cost for the transactions to the function shown in listing 1.[3] This function traverses fsArray in the for loop, and the content of fsArray is obtained from ownerToFashionArray for the given _owner address. The iteration times for the *for* loop is determined by *fsArray.length* (see lines 8 and 5). In addition, the value of *fsArray* is determined by the input *_owner* (line 4). The execution gas cost is related to the iteration times of the function but the latter one can only be decided upon runtime. Thus, there is no way to precisely compute the execution gas cost in only a static way.

C. LEARNING MODELS

To the best of our knowledge, there are no previous studies applying learning algorithms on a gas estimation. In this paper, we define a gas estimation as learning a mapping $f : \mathbb{R}_{\mathbb{N}} \rightarrow \mathbb{R}$, where $\mathbb{R}_{\mathbb{N}}$ is an N-dimension feature, which is a representation of the transaction opcode sequence, and $\mathbb{R}$ is the predicted gas.

The concept of machine learning is to learn a model from existing data with a performance measure metric and give a

[3]This function is excerpted from a contract whose address is 0x163af66AE287EB89554BFd2DE10f7C3Ac9fEDf84.

judgment or prediction on new data. Machine learning algorithms are currently being widely applied to various tasks, including computer vision, natural language processing, and recommendation systems. The feature space and regressor selection are entirely unknown, and there are no widely recognized evaluation metrics for a gas estimation. To solve this challenge, we search for several machine learning and deep learning methods, i.e., random forest, KNN, and SVM, and two different evaluation metrics, the MAPE and the accuracy rate. The performance for each regressor is discussed in Section IV.

Random Forest: A decision tree learning algorithm can build a regression model in a tree structure. It is prone to overfitting when a tree is extremely deep. Thus, a random forest is used to minimize this error. A random forest [11] is a group of decision trees that aggregates their results to one result. Based on the voting strategy, the random forest may produce a better result from assembled models rather than individual models. The random forest model overcomes an overfitting and can be more interpretable because it can explicitly output a weight for each dimension of the features.

K-Nearest Neighbor (KNN): A KNN [12] is a commonly used supervised learning algorithm. Supposing the distance of the samples is defined by a similarity measurement, for example, the Euclidean metric, Minkowski distance, or Manhattan distance, the KNN aims to find the closest K samples from the training set. Based on these K samples, the prediction result is the average with/without the weights of the real output value. KNN methods usually achieve a better performance on datasets with a smaller size.

SVM for regression (SVR): An SVM [13] is a widely used machine learning method, and constructs a margin separator that finds a hyperplane that has the maximum distance between features. The SVM method was initially proposed for classification but can be extended to a regression, called a support vector regression (SVR), although a traditional SVM is based on a linear separable assumption. By defining the inner product of features in terms of a kernel function, the SVM also suits the non-linearly separable problem. Intuitively, a gas estimation is not a linearly separable problem. This inspired us to use the Gaussian Kernel function in our experiments.

LSTM: Because traditional neural networks cannot handle the context information of the time series data, recurrent neural networks are a proposed solution. The information of the history data is preserved by introducing the time-variant hidden state for each network cell, and the relationship between inputs can be learned during a gradient descent. Long short-term memory (LSTM) networks are a modified version of recurrent neural networks, which makes it easy to train by avoiding gradient vanishing and exploding problems. They contain input, remember, and output gates, which gives the LSTM network cells the ability to decide which values to apply and which to abandon. We can treat the input opcode sequence as a time series sequence, in which each opcode can be represented by an embedding word vector. The LSTM aims to give a prediction of the gas based on the new input opcode sequence.

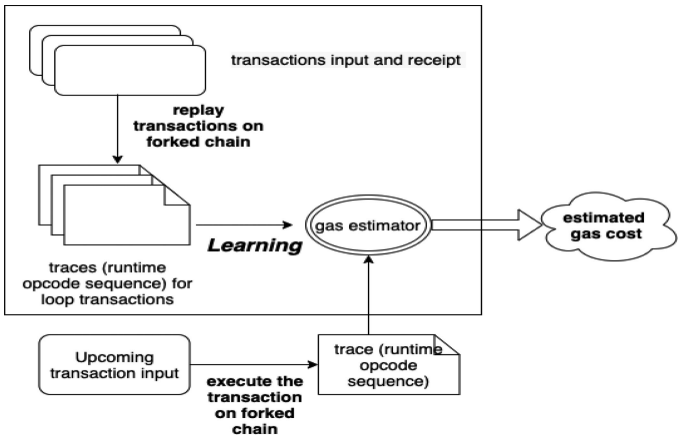


FIGURE 1. Workflow of trace-based approach.

III. OUR APPROACH

Now we will describe the workflow of the proposed approach. There are mainly three steps, as shown in Fig. 1. For simplicity, we refer to the transactions sent to the loop functions as loop transactions. First, we collect the input and receipt for the existing loop transactions. We then replay all loop transactions and extract their trace on the local blockchain. Here, *trace* indicates a transaction executed opcode sequence on an EVM. Finally, we build a gas estimator model based on the transaction trace using machine learning and deep learning algorithms. After a gas estimator construction, for a new loop transaction, we simulate it on a local blockchain to derive its trace and obtain the estimated gas by applying a model to this trace.

A. LOOP TRANSACTION COLLECTION

A loop transaction is the transaction sent to a contract function containing loops. First, for a given contract, we need to select its functions having loops (hereafter called loop functions). Next, we gather existing transactions sent to this contract. By analyzing the inputs for the existing transactions, we collect the transactions sent to the loop functions. Details of the selection and analysis steps are as follows:

1) Select loop functions: We first use Slither [14] to obtain the control flow graph (CFG) for the functions in the contracts. Slither is a static analysis framework that can convert Solidity contracts into an intermediate representation called SlithIR. SlithIR has a node type called an "IF_ LOOP" indicating the start of a loop. In addition, we traverse all functions of the CFGs to collect loop functions that have at least one "IF_ LOOP" node.
2) Gather transactions sent to a contract: Etherscan[4] shows all transaction hashes sent to a contract address. For this study, we searched no more than 2000 recent transaction hashes for considered contracts

[4]https://etherscan.io/

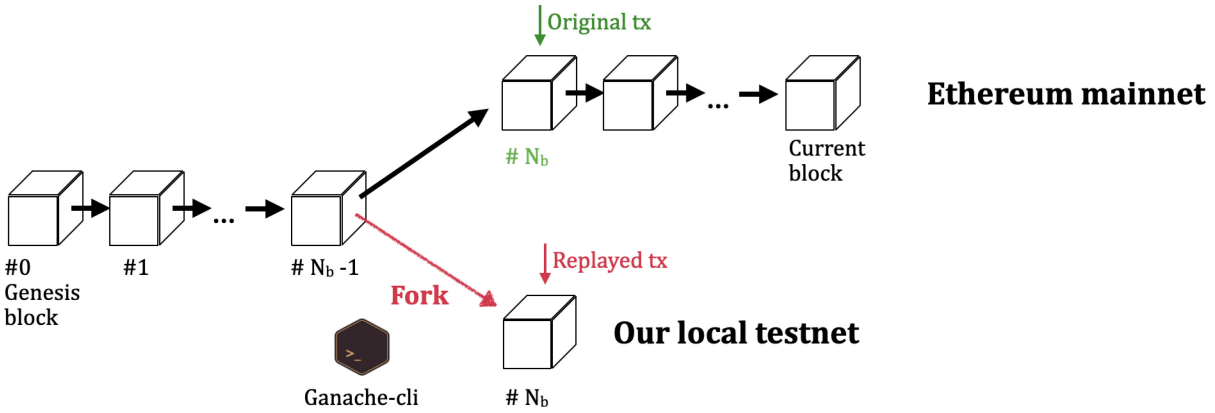


FIGURE 2. Collect runtime trace by replaying a transaction.

from Etherscan. We then pulled the detailed transaction information (e.g., input and transaction sender) from a full node on Ethereum mainnet by calling the *web3.eth.getTransaction()* API. In particular, we deployed a full node based on QuikNode's node service.

3) Analyzed transactions sent to loop functions: The input of a transaction contains the invoked function name and parameters. We use abiDecoder[5] to decode every transaction input to obtain the invoked function name. If the called function name is one of the loop function names, we add this transaction information into our database.

B. TRANSACTION TRACE GENERATION

The transaction trace is the transaction execution opcode sequence on an EVM. A method for generating traces is calling the API *debug.traceTransaction()* [6] from a full node. However, using this API to obtain the trace triggered by a transaction is a slow approach because, for a given transaction hash, it requires finding the previous block where the transaction resides and then replays all preceding transactions before the transaction on the same block [15]. In addition, a further time delay is caused by the remote procedure calls from the API. Chen *et al.* [15] proposed another way to record the traces. They apply a full Ethereum node and replay all transactions during synchronization. After synchronization is finished, the traces are automatically collected. They then aim to collect the traces for all transactions. However, it is costly for us to replay some specific loop transactions using their method.

We propose a new way to obtain a transaction trace, which is illustrated in Fig. 2. Suppose the original transaction is collected in the $\#N_b$ block. We fork Ethereum mainnet on a $\#N_b - 1$ block to start a local testnet. To implement this, we use **Ganache-cli**[7] and a **Infura**[8] node service. Ganache-cli is the command-line version of Ganache. It can be used to build a personal blockchain for development. In particular, it provides a fork command to allow users to fork from another running Ethereum client on the specified block, which allows us to send transactions to contracts residing in mainnet. Infura is a node cluster that frees developers from synchronizing and maintaining an Ethereum node. In our study, an archive node is hosted because it can respond to API requests for any historical blocks. As shown in Fig. 2, the local testnet shares the chain starting from the genesis block to the $\#N_b - 1$ block with Ethereum mainnet. This is applied to construct the same correct state before the original transaction. In particular, we revised the Ethereumjs-VM to collect the trace when replaying the transactions. Ethereumjs-VM is the EVM used in the Ganache blockchain. More concretely, the EVM interpreter executes each opcode on the **runStep** function, and thus we insert the recording code into this function. In this way, when the EVM executes a transaction, the trace is automatically collected.

C. BUILD GAS ESTIMATOR MODELS

After the gas estimator is learned, for a new transaction, we can first execute it on a forked ganache blockchain and derive its trace, and then obtain the estimated gas by inputting its trace into the gas estimator.

However, the traces for the loop transactions are usually long. The maximal trace we collected contains 382,552 opcodes. Some studies [16], [17] on using deep learning algorithms for malware detection based on input opcode sequence have been conducted. These approaches only take the first L opcodes to meet the need for a deep learning network of the unified input length. As observed, the larger the L is, the more memory and computation time required to train the neural network. For a gas estimation, we cannot simply follow this rule because each opcode contributes to the final predicted gas. In addition, in our experiments, a memory overflow error is raised owing to the long sequence, even with a batch size of one. It is difficult to feed this long sequence into any existing learning models directly.

In our previous study [8], we extracted the opcode frequency and dynamic opcode sequence from the transaction trace. The frequency-based method extracts the frequency of

[5]https://github.com/ConsenSys/abi-decoder
[6]https://github.com/ethereum/go-ethereum/wiki/Management-APIs
[7]https://github.com/trufflesuite/ganache-cli
[8]https://infura.io/

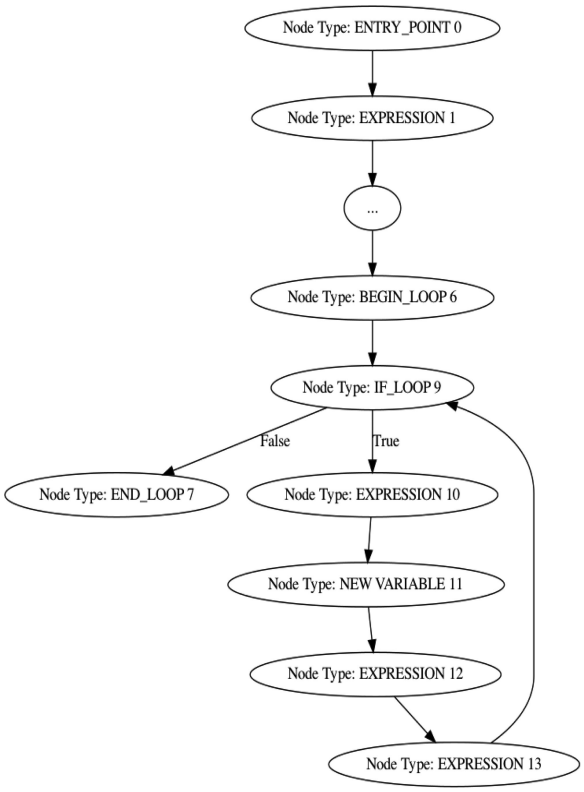


FIGURE 3. **Control flow graph for function in listing 1.**

all opcodes and feeds it to different learning models. The sequence-based method only considers a dynamic opcode sequence but maintains the original opcode order. In this study, in addition to mentioned two types of features, i.e., the opcode frequency and dynamic opcode sequence, we combine the opcode frequency and *function vector* as the third type of features for a transaction. The function vector describes the number of occurrences of different structural characteristics in the function. As shown in Fig. 4, we attach the loop function vector at the end of opcode frequency. In the next three subsections, we describe the details for the three types of features.

1) OPCODE FREQUENCY

There are 141 opcodes used on an EVM. In Fig. 4, the frequency-based method is used to extract the frequency of all opcodes and feed it to the different learning models. The sequence shaded in blue is the frequency of all opcodes extracted from the original opcode sequence on the left. For example, the value of 612 in position 2 shows that the opcode ADD occurred 612 times in the original trace (opcode sequence). For opcode frequency features, we consider three supervised machine learning models, namely, random forest, KNN, and SVR.

2) COMBINATION OF OPCODE FREQUENCY AND LOOP FUNCTION VECTOR

The loop function vector describes the number of occurrences for different structural characteristics in the loop function. We add the loop function vector to the end of the opcode frequency table to build a new type of feature, as shown in Fig. 4.

The loop function vector is constructed in four steps. First, we use Slither [14] to obtain the control flow graph (CFG) for a loop function. Second, we modify the Exas algorithm [18] to extract features of two types of patterns (n_path_pattern and node_pattern) from the control flow graph of a function.

1) The n_path_pattern contains some directed paths that contain n nodes. For any two nodes in the path, one node can be reached from another node by a directed edge. The feature for an n_path_pattern is a sequence of node types along the path. For example, for the graph 3, a feature of 3_path_pattern is "Node Type: BEGIN LOOP 6 → Node Type: IF LOOP 9 → Node Type: EXPRESSION 10".
2) The node_pattern associates a node with p incoming and q outgoing edges. For example, node 9 in Fig. 3 has two incoming edges from nodes 6 and 13, and two outgoing edges to nodes 7 and 10. Thus, the features for node 9 is "Node Type: IF_LOOP 9 → 2 → 2".

We partly modified the vector computing algorithm in [18] to extract features for a loop function, as shown in Algorithm 1. The belief is that the features for a graph can be computed from the features of its sub-graphs. From an empty graph with only one node, we add edges to the graph one by one. For an edge (u,v) and existing graph G, we add new features in terms of whether u or v exists in G.

Third, we have to collect all features for all loop functions (Algorithm 2). Fourth, for a contract function, its vector is computed as the occurrence counts of its features among all features (Algorithm 3).

Let us consider the function in listing 1 to understand the construction of a loop function vector. The parts of the control flow graph for this function are shown in Fig. 3. Each node stands for a basic block consisting of expressions. For example, the expression in "Node Type: EXPRESSION 1" is as follows:

```
1 require(bool)(_owner != address(0))
```

Using Algorithm 1, some of the features extracted from the control flow graph in Fig. 3 are shown in Table 1. We collected 1124 features from all loop functions using Algorithm 2. Thus, the *loop function vector* is a 1124-dimensional vector. As shown in Fig. 4, supposing the transaction relating to the original opcode sequence on the left is sent to a function f, then the vector shaded in green on the right is the function vector. The value in position 2 indicates the number of occurrences (4) of the second feature (Node Type: EXPRESSION: 16→ 1→1).

3) DYNAMIC OPCODES SEQUENCE

We checked the go-ethereum[9] source code and divided it into three classes: constant cost opcodes, dynamic cost opcodes, and both constant and dynamic cost opcodes, as shown in

[9]https://github.com/ethereum/go-ethereum

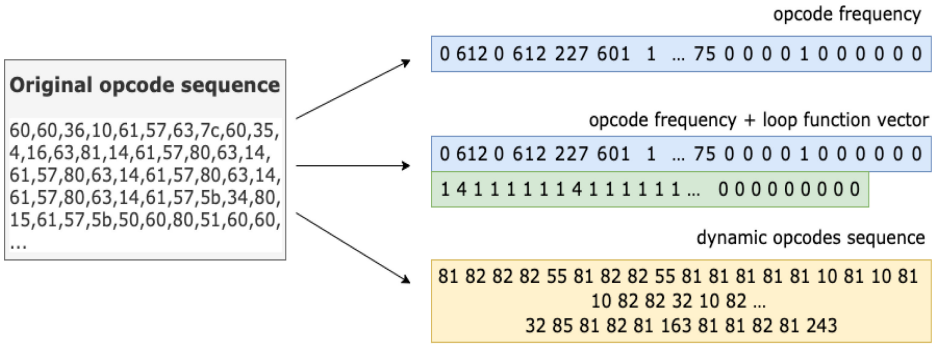


FIGURE 4. Three types of features for a transaction.

Table 2. An EVM charges 117 opcodes and 10 opcodes in terms of constant and dynamic gas costs, respectively. For example, an **ADD** opcode has a gas cost of three, and an **EXP** gas cost can only be decided at runtime. In addition to a constant gas cost, 14 opcodes also have dynamic gas, such as **SHA3**, which has a fixed gas cost of 30 and a dynamic cost relating to memoryGasCost.

We propose a sequence-based method that only contains a dynamic opcode sequence while maintaining the original opcode order. In Fig. 4, the sequence shaded in yellow is the dynamic opcodes extracted from the original opcode sequence on the left. Because only 24 opcodes have a dynamic gas cost, the dynamic opcode sequences shorten the original trace.

IV. RESULTS

We use the Smartbugs [19] contract dataset containing 47,398 unique contracts. As stated in Section III, for each contract program, we employ Slither [14] to select its loop functions, i.e., functions that contain at least one loop. We observed that 10,855 contracts have loop functions, which is 23% of all Smartbugs contracts. We searched the recent 2000 transaction records of the 10,855 contracts from Etherscan and analyzed the transaction inputs. The results show that there are 706 contracts with transactions sent to the loop functions. Up to 50 transactions are sent to each of the 457 loop contracts, which amounts to 64.7% of all loop contracts. In addition, 64 loop contracts range in loop transactions of 500 to 2000, which occupy 9.1% of all loop contracts. Almost a quarter of the contracts have loop functions, although users do not often send transactions to them. The reasons behind this might be as follows: First, smart contracts might contain loop-related vulnerabilities, such as an unbounded loop [3], despite there being no effective tool that can remedy them. Second, there is no practical tool to estimate the gas cost for the loop functions.

As stated in Section III, we need to replay the loop transactions on a forked testnet and collect their transaction traces. In our experiments, the average transaction replay time is approximately 30 s. In addition, 3 min is required to replay an extremely complicated transaction. Considering the time limits, we replayed the recent *up to 10* transactions[10] for each loop contract. We collect traces for 5718 transactions. The opcode length for these traces ranges from 43 to 382,552.

To evaluate the effectiveness of our approach, we consider the following two metrics:

1) Mean Absolute Percentage Error (MAPE): This expresses an error as a ratio defined in the formula $L = \frac{1}{n}\sum_{i=1}^{n}|\frac{g^i_{actual}-g^i_{pred}}{g^i_{actual}}| * 100$, i.e., the average difference between a predicted gas and an actual gas is divided by the actual gas, where the predicted gas is directly estimated by the learned gas estimator. A smaller MAPE indicates a better prediction performance.
2) Prediction accuracy rate: Because the learned estimator may underestimate the gas for certain transactions, we compute this metric as different accuracy rates by adding an additional gas to the estimator provided gas. Here, the accuracy indicates whether the predicted gas is higher than actual gas. For example, in Fig. ??, the prediction is accurate for almost 45% of the transactions when directly using the estimator. However, when adding 5000 to the estimator predicted gas, the prediction is accurate for over 70% of the transactions.

A. OPCODE FREQUENCY BASED METHOD PERFORMANCE

The frequency-based method fixes the opcode frequency to 141 as a feature. For this method, in addition to collecting 5718 transactions, we replayed the transactions for the four representative hash contracts listed in Table 3. We first counted the opcode frequency for each trace, and then applied machine learning models (random forest, KNN, and SVR) to the frequency vectors separately. The training and testing sets were randomly split at 70% and 30%, respectively. The training time is less than 5 s. The MAPE results are shown in Table 3. We have three observations:

1) In general, a gas estimation based on transactions to the same contract has a lower error rate than transactions for different contracts. For example, if we use a random forest learning algorithm to estimate the gas for the transactions to contract 0x92240..., and to combine four

[10]These 10 transactions invoke the same loop function.

Algorithm 1: get_function_feature (Adapted From [18]).

Input: The source code of a contract C, loop function name f
Output: All features for the contract

```
Construct a control flow graph G for f from C
edges = G.edges, nodes = G.nodes
Initialize a new graph G', G'.nodes = [nodes[0]]
node_pattern[nodes(0)] = [nodes(0).label, 0, 0]
1_path_pattern = [nodes(0).label]
Initialize 2_path_pattern, 3_path_pattern, and
  4_path_pattern with empty list
while edges is not empty do
    edge = edges[0], u = edge[0], v = edge[1]
    if u is not in G'.nodes and v is in G'.nodes then
        add u to G'.nodes, add (u, v) to G'.edges
        node_pattern[u] = [u.label, 0, 1]
        node_pattern[v][1] ++
        add u.label to 1_path_pattern
        add (u.label, v.label) to 2_path_pattern
        for each path in i_path_patterns do
            if path starts from v then
                add u + path to (i+1)_path_patterns
        end
        delete edge from edges
    else if u is in G'.nodes and v is not in G'.nodes
      then
        add v to G'.nodes, add (u, v) to G'.edges
        node_pattern[v] = [u.label, 1, 0]
        node_pattern[u][2] ++
        add v.label to 1_path_pattern
        add (u.label, v.label) to 2_path_pattern
        for each path in i_path_patterns do
            if path ends in u then
                add path + v to (i+1)_path_patterns
        end
        delete edge from edges
    else if u is in G'.nodes and v is in G'.nodes then
        add (u, v) to G'.edges
        node_pattern[v][1] ++
        node_pattern[u][2] ++
        add (u.label, v.label) to 2_path_pattern
        collect paths_end_in_u
        collect paths_start_from_v
        for path1 in paths_end_in_u do
            for path2 in paths_start_from_v do
                add path1+path2 to (len(path1) +
                  len(path2))_path_patterns
            end
        end
        delete edge from edges
    else
        move edge to the end of edges
    end
end
Initialize features with empty list
add all elements from node_pattern and i_ path_pattern
  (i can be 1,2,3,4) to features
return features
```

Algorithm 2: get_all_features.

Input: The source codes of all contracts C_{all} and their respective loop function names
Output: Contract feature collection

```
initialize features_collection with empty list
foreach contract source code C, loop function name f
  do
    feature = get_contract_feature(C, f) (Algorithm 1)
    if feature does not exist in features_collection then
        add feature to features_collection
    end
end
return features_collection
```

Algorithm 3: get_function_vector.

Input: The source code of a contract **C**, loop function name **f**, **features_collection** from **Algorithm 2**
Output: The vector **V** for the contract

```
contract_features = get_contract_feature(C, f)
initialize feature_and_counts as empty list
foreach feature in contract_features do
    if (feature,_) does not exist in feature_and_counts
      then
        T = occurrence times of the feature in
          contract_features
        feature_and_counts[feature] = T
    end
end
initialize a mapping M from feature to counts
foreach feature in features_collection do
    M[feature] = 0
end
foreach feature in feature_and_counts do
    M[feature] = feature_and_counts[feature]
end
initialize a vector list V
foreach feature in features_collection do
    add M[feature] to V
end
return V
```

contract transactions separately, the former MAPE is 0.78, and the latter is 1.71.

2) In most cases, random forest and KNN can have a lower error rate than the SVR. Consider the contract 0x117cb..., the MAPE for a random forest and a KNN are 5.05 and 6.94, respectively, which is lower than the 9.74 predicted by the SVR.
3) Recall that we replayed less than 10 recent transactions for each loop contract and collected 5718 traces. The MAPE for these transactions is distinctly higher than that combined for four contract transactions.

To further validate whether a random forest is the most suitable model, we collect transactions for more loop contracts. The number of loop transactions ranges from 12 to 121. Fig. 6

TABLE 1. The Patterns and Features Extracted From Fig. 3

Pattern	Features of control flow graph			
1_path_pattern	Node Type: ENTRY_POINT 0	Node Type: IF_LOOP	Node Type: EXPRESSION. 12	...
2_path_pattern	Node Type: ENTRY_POINT 0 -> Node Type: EXPRESSION 1	Node Type: EXPRESSION 10 -> Node Type: NEW VARIABLE 11	Node Type: EXPRESSION 12 -> Node Type: EXPRESSION 13	...
3_path_pattern	Node Type: BEGIN_LOOP 6 -> Node Type: IF_LOOP 9 -> Node Type: EXPRESSION 10	Node Type: BEGIN_LOOP 6 -> Node Type: IF_LOOP 9 -> Node Type: END_LOOP 7	Node Type: EXPRESSION 12 -> Node Type: EXPRESSION 13 -> Node Type: IF_LOOP 9	...
4_path_pattern	Node Type: BEGIN_LOOP 6 -> Node Type: IF_LOOP 9 -> Node Type: EXPRESSION 10 -> Node Type: NEW VARIABLE 11	Node Type: IF_LOOP 9 -> Node Type: EXPRESSION 10 -> Node Type: NEW VARIABLE 11 -> Node Type: EXPRESSION 12	Node Type: NEW VARIABLE 11 -> Node Type: EXPRESSION 12 -> Node Type: EXPRESSION 13 -> Node Type: IF_LOOP 9	...
node_pattern	Node Type: ENTRY_POINT 0 ->0 ->1	Node Type: IF_LOOP 9 ->2 ->2	Node Type: EXPRESSION 13 ->1 ->1	...

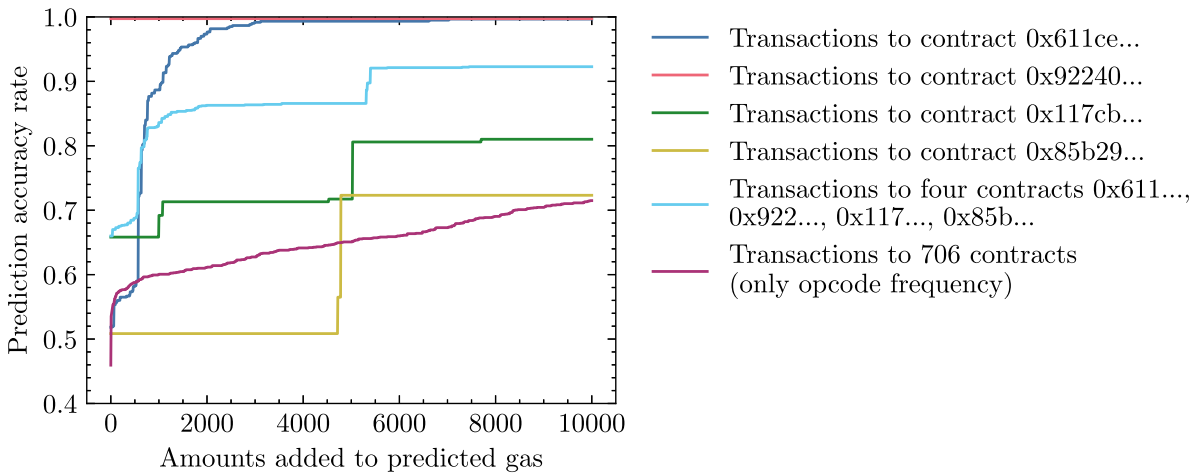


FIGURE 5. Prediction accuracy rate for transactions to different contracts.

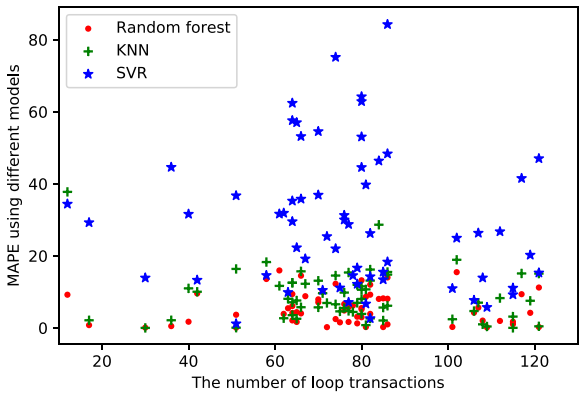


FIGURE 6. MAPE for different numbers of loop transactions.

depicts the MAPE results using three models in different numbers of loop transactions. We apply the same inspection as our previous study [8], i.e., the MAPEs for a random forest are generally lower than that for a KNN and an SVR.

Fig 5 list the prediction accuracy rate with an incremented gas using a random forest algorithm for the six types of transactions listed in Table 3. Generally, more gas that is added to the estimator provided gas, the higher the number of transactions that occur with the increased gas over than the actual gas. For example, for transactions to contract 0x117cb..., as shown in the green line of Fig 5, if we add 2000 to the predicted gas from the gas estimator, the prediction accuracy rate can reach 70%, i.e., for 70% of the tested transactions, we can make sure that incremented gas is higher than the actual gas. However, if

TABLE 2. Opcodes and Their Gas Cost

Opcodes number	Opcodes	Constant gas	Dynamic gas	Percentage in all opcodes
117	STOP, ADD, MUL, SUB, DIV, SHL, CALLVALUE,...	√		83%
14	SHA3, CALLDATACOPY, CODECOPY, MLOAD, CREATE, CALL,...	√	√	10%
10	EXP, SSTORE, LOG0, RETURN, REVERT, SELFDESTRUCT,...		√	7%

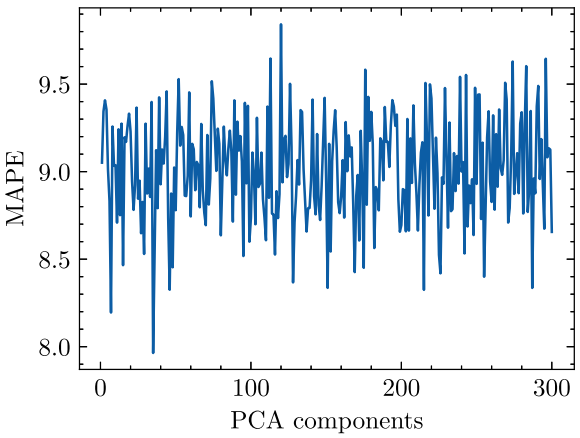


FIGURE 7. MAPE for different numbers of PCA components.

we add 6000 to the predicted gas from the gas estimator, the prediction accuracy rate can reach 82%.

B. COMBINATION OF OPCODE FREQUENCY AND LOOP FUNCTION VECTOR BASED METHOD PERFORMANCE

As with the frequency-based method only, we chose 5718 transaction for 706 loop contracts as our dataset. For each transaction, if the transaction calls the function f, we first compute the function vector V for f and attach V to the end of the opcode frequency features for the transaction. The dimensions for V and the opcode frequency are 1124 and 141, respectively. We applied a principal component analysis (PCA) technique [20] to reduce the dimensionality of V. The training and testing sets were randomly split at 70% and 30%, respectively. The training time is less than 5 s. In Table 4, we observe the following:

1) For the same number of PCA components, a random forest model provides the smallest MAPE results.
2) For the random forest model, the method that appends the function vector to the opcode frequency can decrease the MAPE compared with that having only the opcode frequency. For example, using only the opcode frequency, the MAPE for a random forest is 9.95. However, the subsequent numbers in the same column are all lower than 9.95.

Fig. 7 shows the MAPEs using PCA components ranging from 1 to 300 in number. The maximum MAPE is still smaller than 9.95 (i.e., MAPE in the random forest for using only the opcode frequency), which shows that the function vector can help make the estimated gas closer to the actual gas.

C. DYNAMIC OPCODE SEQUENCE BASED METHOD PERFORMANCE

The sequence-based method maintains a dynamic opcode sequence as features, the maximal length of which is 14,267. For the sequence-based method, we chose 5718 transactions for 706 loop contracts as our dataset. We first extracted dynamic sequences (i.e., a sequence only containing dynamic opcodes) from each trace and fed these dynamic traces to the LSTM models. The training and testing sets were randomly split at 70% and 30%, respectively. The training time is approximately 3 days. The MAPE for LSTM is over 800, which is far higher than the MAPE for the random forest, KNN, and SVR.

D. EVALUATION OF OUR METHODS

First, our methods are effective in estimating the gas costs for loop transactions. A dynamic sequence is unsuitable for a gas estimation because the predicted gas is significantly different from the actual gas. The combination of the opcode frequency and function vector can better describe the original trace than only the opcode frequency and dynamic sequence. In addition, for the same features, the random forest and KNN have a better estimation rate than the SVR. Moreover, a prediction of the transactions from the same contract is better than that from different contracts.

Second, our method is robust. The smart contracts used in our experiments are all crawled from Ethereum mainnet, and we assume that this contract dataset have covered most types of contracts. So our method is applicable for gas estimation of more transactions to other new contracts. Besides, we can get a better prediction accuracy rate with more transaction traces.

E. LIMITATION

1) As shown in Fig 2, we assume that the Ethereum state on block $\#N_b - 1$ is the correct state before the execution of the original transaction. Suppose the replayed transaction is sent to contract C. Here, we consider that the preceding transactions in block $\#N_b$ do not change the state of contract C. To mitigate this, we will try to analyze the relationships among transactions in the same block.
2) The gas relating the runtime trace sequence is the EVM execution gas cost, which is not the transaction gas cost. The gas cost of a specific transaction contains three parts [1]: the intrinsic gas cost, execution gas cost, and refund gas cost. The intrinsic gas includes a fixed message call transaction fee of 21,000 and the gas for the associated data of the transaction. We ignore the intrinsic gas cost differences for different transactions because gas cost for the transaction data is far lower than the overall transaction gas cost. In addition, we assume that the refund gas for all loop transactions is zero. However,

TABLE 3. MAPE Results Using a Random Forest, a KNN, and an SVR

Contract hash	Loop transactions number	MAPE		
		Random forest	KNN	SVR
0x611ce695290729805e138c9c14dbddf132e76de3	2000	1.40	1.49	17.46
0x9224016462b204c57eb70e1d69652f60bcaf53a8	1238	0.78	0.59	0.96
0x117cb292e97a593fbca38b5cd60ec7144d4ca8c9	790	5.05	6.94	9.74
0x85b2949cea65add49c69dac77fb052596bc5ddd4	590	1.49	1.49	19.01
Combine above four contracts transactions	4618	1.71	2.08	53.38
706 contracts hash (only opcode frequency)	5718	9.95	17.83	67.19

TABLE 4. MAPE Results for Only Opcode Frequency and Both Opcode Frequency and Function Vector

	PCA components number	MAPE		
		Random forest	KNN	SVR
Opcode frequency		9.95	17.83	67.19
Opcode frequency + function vector	35	8.74	19.65	67.33
	120	9.45	17.51	67.57
	214	9.01	17.53	67.44
	286	9.75	17.84	67.74

our method can be used (although not perfectly) even when considering the intrinsic and refund gas costs. Later, we will study the impact of the intrinsic gas and refund gas in the overall transaction gas cost.

V. RELATED STUDIES

Gas estimation: Albert *et al.* constructed a gas analyzer called GASOL [7], which can over-approximate the gas consumption of a function. In addition, Marescotti *et al.* [6] presented two methods for deciding the exact worst-case gas consumption. Signer provided Visualgas [5], a tool to visualize how gas costs relate to different parts of the code. However, the actual transaction gas costs on mainnet have not been compared with the predicted costs to prove the effectiveness of these methods. Based on feedback-directed mutational fuzzing, Ma *et al.* designed GasFuzz to construct inputs that maximize the gas cost [4]. The Ethereum community has also developed tools to help estimate such costs. For example, Solc[11] statically predicts the gas cost, and Remix[12] provides a debugger to list the gas cost of a transaction execution trace. Once users upload the contract codes to Remix, Remix will estimate the gas costs after compilation. Both Solc and Remix show an infinite gas cost for any transaction calling functions with loops. Moreover, the Web3[13] package can apply a gas estimation by executing the transaction directly in the EVM of the Ethereum node, although this only makes sense when this transaction does not throw any exceptions. Paterno *et al.* presented a tool called a gas exactimation [21], which addresses the issue in EIP-144 by exploring how the gas held at any nested stack depth/frame influences the gas outside of its execution context.

Gas optimization and vulnerability detection: Chen *et al.* identified seven gas cost patterns for the Solidity code and developed GASPER [23] to locate three of these patterns by analyzing the bytecodes. They later listed 24 anti-patterns and implemented GasReducer [24] to detect and replace them with an efficient code. They focused on optimizing the gas usage whereas we want to apply a gas estimation. Nagele *et al.* applied the super-optimization idea [25] to a gas optimization. They encoded a sequence of instructions within a basic block as SMT formulas and found cheaper bytecodes using a constraint solver. Albert *et al.* [26] also employed a super-optimization approach to optimize the instruction sequence. They extracted a stack functional specification from a block and synthesized a new optimized block with the minimal gas cost, which has the same stack functional specification as the extracted version. Grech *et al.* [3] surveyed three gas-related vulnerabilities and detected them at the bytecode level.

VI. CONCLUSION

In this work, we identified the importance of estimating the gas costs for transactions sent to loop functions. We proposed a trace-based approach to estimate the transaction gas. We abstracted the original trace for three types of features: (1) the opcode frequency only, (2) a combination of the opcode frequency and a function vector, and (3) a dynamic opcode sequence. We applied three machine learning models (i.e., random forest, KNN, and SVR) in features (1) and (2) and LSTM in feature (3). The results show that our method is

[11] https://github.com/ethereum/solidity
[12] https://github.com/ethereum/remix
[13] https://web3js.readthedocs.io/en/v1.2.0/web3-eth.html

effective in estimating the gas cost for loop functions. In particular, random forest and a KNN have a better estimation rate than an SVR and an LSTM. In addition, we provide a dataset that contains 5718 traces for transactions to the loop functions. The dataset suggests that more research is needed to estimate the gas cost for a loop transaction.

In the future, to improve the prediction accuracy rate, we plan to collect more transaction traces to train our gas estimation model and extract new features from the original trace. Besides, we would like to apply our idea to estimate the gas costs for functions other than loops.

REFERENCES

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[2] C. Liu, J. Gao, Y. Li, and Z. Chen, "Understanding out of gas exceptions on ethereum," in *Proc. Int. Conf. Blockchain Trustworthy Syst.*, Berlin, Germany; Springer, 2019, pp. 505–519.

[3] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *in Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018, pp. 1–27.

[4] F. Ma *et al.*, "Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability," 2019, *arXiv:1910.02945*.

[5] C. Signer, "Gas cost analysis for ethereum smart contracts," Master's thesis, Dept. Comput. Sci., ETH Zurich, 2018.

[6] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, and N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*, Berlin, Germany: Springer, 2018, pp. 450–465.

[7] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas analysis and optimization for ethereum smart contracts," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2020, pp. 118–125.

[8] C. Li, S. Nie, Y. Cao, Y. Yu, and Z. Hu, "Dynamic gas estimation of loops using machine learning," in *Proc. Int. Conf. Blockchain Trustworthy Syst.*, Berlin, Germany; Springer, 2020, pp. 428–441.

[9] R. S. Boyer and J. S. Moore, "A mechanical proof of the unsolvability of the halting problem," *J. ACM (JACM)*, vol. 31, no. 3, pp. 441–458, 1984.

[10] Y. Fu *et al.*, "Evmfuzzer: detect evm vulnerabilities via fuzz testing," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 1110–1114.

[11] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[12] L. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, 2009.

[13] e. a. Bernhard E. Boser, "A training algorithm for optimal margin classifiers," 2010.

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

[15] T. Chen *et al.*, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1503–1520.

[16] I. Santos, F. Brezo, B. Sanz, C. Laorden, and P. G. Bringas, "Using opcode sequences in single-class learning to detect unknown malware," *IET Inf. Secur.*, vol. 5, no. 4, pp. 220–227, 2011.

[17] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Android malware detection using deep learning on api method sequences," 2017 *arXiv:1712.08996*.

[18] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, Berlin, Germany: Springer, 2009, pp. 440–455.

[19] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 530–541.

[20] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Trans. Royal Soc. A: Math., Phys. Eng. Sci.*, vol. 374, no. 2065, 2016, Art. no. 20150202.

[21] "gas-exactimation," https://www.trufflesuite.com/blog/ethereum-gas-exactimation.

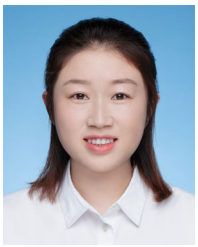
[22] "eth-gas-reporter," https://www.npmjs.com/package/eth-gas-reporter.

[23] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering*, 2017, pp. 442–446.

[24] T. Chen *et al.*, "Towards saving money in using smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.: New Ideas and Emerg. Technol. Results*, 2018, pp. 81–84.

[25] J. Nagele and M. A. Schett, "Blockchain superoptimizer," 2020 *arXiv:2005.05912*.

[26] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of superoptimized smart contracts using max-SMT," in *Proc. Int. Conf. Comput. Aided Verification*, Berlin, Germany: Springer, 2020, pp. 177–200.



CHUNMIAO LI received the bachelor's degree from Northwest University in 2015 and the master's degree from Shanghai Jiao Tong University in 2018. She is currently the Ph.D. student with the Graduate University for Advanced Studies (SOKENDAI), Japan. Her current research interests include blockchain, smart contract security, and program analysis.

SHIJIE NIE received the Ph.D. degree from the Graduate University for Advanced Studies (SOKENDAI), Japan. He is currently a Project Researcher with the National Institute of Informatics (NII), Japan. His current research interests include computational photography and deep learning.

YANG CAO received the Ph.D. degree from the Kyoto University in 2017. He is an Assistant Professor with Yoshikawa & Ma Lab, Department of Social Informatics, Kyoto University. His research interests include data privacy and security.

YIJUN YU received the B.Sc., M.Sc., and Ph.D. degrees from the Department of Computer Science at Fudan University in 1992, 1995, 1998, respectively. He is a Senior Lecturer with the Department of Computing and Communications in The Open University. His research interests include the areas of requirements engineering, maintenance and evolution, security and privacy. He is a member of the IEEE Computer Society and the British Computer Society.

ZHENJIANG HU (Fellow, IEEE) received the B.S. and M.S. degrees from Department of Computer Science and Engineering of Shanghai Jiaotong University in 1988 and 1991 respectively, and Ph.D. degree from Department of Information Engineering of University of Tokyo in 1996. He is a Professor and the Chair with the Department of Computer Science and Technology, Peking University. He is Fellow of JFES (Japan Federation of Engineering Society, 2016), ACM Distinguished Scientist (2016), Member of Europen Academy of Science (2019). His research interest include programming languages and software engineering in general, and functional programming, bidirectional transformation, and software adaptation in particular.