Dynamic Gas Estimation of Loops Using Machine Learning

Chunmiao Li[1,2(✉)], Shijie Nie[1], Yang Cao[3], Yijun Yu[4], and Zhenjiang Hu[5]

[1] National Institute of Informatics, Tokyo, Japan
{chunmiaoli1993,nsj}@nii.ac.jp
[2] SOKENDAI (The Graduate University for Advanced Studies), Hayama, Japan
[3] Kyoto University, Kyoto, Japan
[4] Open University, Milton Keynes, UK
[5] Peking University, Beijing, China

Abstract. Smart contracts on Ethereum can encode business logic and have been applied to many areas, such as token exchange, games, and others. Unlike general programs, the computations of contracts on Ethereum are restricted by the gas limit. If a transaction runs out of gas limit before execution finishes, EVM throws an out-of-gas (OG) exception, and the entire transaction fails, which reverts state before the transaction starts, but transaction fee is still deducted. It is essential to do gas estimation before sending transactions. Existing works mostly fail in estimating gas for loop functions because the iteration times of loops can not be statically decided. But we found that a quarter of all contracts have loop functions, and gas cost for loops is higher than for other functions. So it is necessary to do gas estimation for loop functions.

In this work, we propose a gas estimation approach based on transaction trace to estimate gas for loop functions dynamically. Our main idea is that we can learn the relationship between historical transactions traces and their gas to estimate gas for new transactions. We implement our approach in machine learning algorithms. The results show that random forest and K-nearest neighbors can achieve a better estimation accuracy rate than SVR and LSTM.

Keywords: Ethereum · Smart contracts · Gas estimation · Out-of-gas · Machine learning

1 Introduction

Ethereum [17] is the most popular public blockchain now, not only because it provides a decentralized, shared ledger, which allows all users to participant in the ledger update activities, but also it builds a "world computer" which can host and execute programs. These programs are so-called smart contracts [17]. 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. Then these Solidity programs are compiled to bytecode and stored on the code field of a newly built contract account after

Z. Zheng et al. (Eds.): BlockSys 2020, CCIS 1267, pp. 428–441, 2020.
https://doi.org/10.1007/978-981-15-9213-3_34

the contract creation transaction succeeds. One can send a transaction to the contract account when he wants to call a function on the contract. Once all Ethereum nodes verify the transaction, the Ethereum Virtual Machine (hereafter EVM) will run contract runtime bytecode on the transaction input data.

EVM is a stack-based, Turing-complete machine that can program any computation that a Turing Machine can execute, such as loops. To prevent resource waste due to infinite loop and make sure that contract programs can stop somewhere, the computation effort to execute EVM instructions are charged in the unit of gas. When a user sends a transaction, he needs to specify a gas limit attached to the transaction. *GasLimit* is the maximum available gas amount for transaction execution. But if transaction execution needs more gas than gasLimit, EVM will emit an out-of-gas exception immediately, and all executed operations are reverted. Meanwhile, the expenses for purchasing the gas limit are transferred to beneficiary accounts. The out-of-gas exception accounts for over 90% among all exceptions on Ethereum and causes substantial financial losses [11]. The leading root causes [11] for this exception are: 1. users are not familiar with the transaction execution mechanism; 2. there is no useful tool for gas estimation. There are mainly two ways to prevent out-of-gas exceptions. One way is to detect contracts with gas-focused vulnerabilities [9] to prevent users from calling vulnerable functions. The other one does gas estimation, i.e., given a transaction, we need to estimate its gas cost. Some work has devoted to gas estimation [1,12,13,16]. For example, Solc[1] statically predicts the gas cost for all contract functions. Marescotti et al. [13] apply symbolic model checking methods to detect the worst-case gas cost.

We found that a quarter of all contracts have loop functions, and gas costs for loops are higher than for other functions. So it is necessary to do gas estimation for loop functions. Unfortunately, existing methods mostly fail in estimating the gas cost for transactions to loop functions (i.e., functions containing loops). Static analysis cannot figure out the iteration times of any loops so that Gasol [1], a gas estimation tool, fails when the maximal number of iteration times is unbounded. Dynamic methods often send transactions to local testnet and observe the gas cost, but this gas is not the same as the actual transaction gas cost because the Ethereum mainnet may change and differs from the testnet.

In this work, we propose a novel approach to dynamically estimate gas for loop functions based on transaction execution trace. The insight is that gas costs for new transactions can be predicted based on analyzing history transactions. Our main idea is to learn the relationship between transaction trace and gas from historical transactions and apply this to gas estimation for new transactions. We consider using machine learning algorithms to determine these relations. As far as we know, we are the *first* to introduce machine learning ideas to gas estimation. But it is nontrivial to implement this idea because of two challenges: (1) how to collect traces for a lot of specific historical transactions; (2) traces for loop transactions are very long, and the longest trace we observed is 382, 552. It is hard to feed this long sequence to any existing learning models directly.

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

To address the challenge (1), we instrument Ethereum-js virtual machine[2] to automatically record trace when replaying historical transactions in the forked chain. For challenge (2), we take two abstractions of the trace as features and feed them into different learning models. The first abstraction is frequency for 141 opcodes used on EVM, which are input to three learning models: random forest, K-nearest neighbors (KNN), and SVM for regression (SVR). EVM charges dynamic gas for 24 opcodes depend on runtime state. The second abstraction is dynamic opcodes sequence, which is sent to a Long Short-Term Memory (LSTM) model. The experimental results show that our approach is effective in gas estimation for loop functions. The Mean Absolute Percentage Error (MAPE) ranges from 0.59 to 168.77 in different learning algorithms. Generally, the random forest and KNN can achieve a better prediction accuracy rate than SVR and LSTM.

In summary, our contributions are list as follows.

- We provide a novel approach to estimate gas based on transaction execution trace. The main idea is that the relationship between transaction trace and gas from historical transactions can be learned to estimate gas for new transactions. As far as we know, we are the *first* using machine learning for gas estimation.
- We consider the random forest, K-nearest neighbors (KNN), SVM for regression (SVR), and LSTM learning models in our experiments. The results show that the random forest and KNN can achieve a better prediction accuracy rate than SVR and LSTM.
- We provide a dataset contain opcodes execution sequence and gas costs for 5718 transactions specially sent to loop functions. This dataset can be used for later research on studying the gas cost of transactions to loop functions.

2 Preliminary

2.1 Gas Mechanism on Smart Contracts

Blockchain is a decentralized, shared ledger, and Ethereum [17] is the most popular public blockchain now. 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) to encode business logic. Once a user sends a transaction to a contract account, the contract *opcode* will be executed in Ethereum Virtual Machine (hereafter EVM). Given a transaction, the executional *opcode sequence* in EVM is called a *transaction trace*. All transactions need to be performed 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 in Ethereum are subject to fees [17]. The fee is measured by units of gas.

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,

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

the available gas is reduced by executing opcodes. Suppose the gas limit is G_1 and transaction actual execution cost 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 relationships among G_1, G_2 and G_b, the different transaction execution scenarios are given below:

1. $G_2 < G_b$ and $G_1 >= G_2$: The transaction can be included to a block and succeeds. The gas remained 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 to a block but fails with error. The EVM will emit an out-of-gas exception since there is no available gas to support further operations during the transaction execution. At this time, all gas cost is delivered to miner's account (beneficiary account), and all states done 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 big G_1 is. For example, consider the contract function below. The gas requirements for executing `batchAirDrop` function is related to the length of transaction input `recipients`. The loop iteration times are decided by user input so the maximal gas cost for the loop is unbounded. If the length is too long, the gas cost to execute the loop function might become so huge that it exceeds the block gas limit, and this transaction will not be included in any block. In other words, no matter how much a user could afford, this transaction will always fail.

```
function batchAirDrop(address token_address,
             address[] recipients, uint256 ncash) {
    AToken token = AToken(token_address);
    for(uint i = 0 ; i < recipients.length ; i++) {
        address recipient = recipients[i];
        require(token.transfer(recipient, ncash));
    }
}
```

2.2 Learning Models

As far as we know, there is no previous work that applied learning algorithms on gas estimation. In this paper, we define gas estimation as learning a mapping $f : \mathbb{R}_{\mathbb{N}} \rightarrow \mathbb{R}$, where $\mathbb{R}_{\mathbb{N}}$ is N-dimension features, which is a representation of the transaction opcode sequence, and $\mathbb{R}$ is the predicted gas value.

The concept of machine learning is to learn a model from existing data with a performance measure metric and give a judgment or predictions on new data. Nowadays, machine learning algorithms are 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 is no widely recognized evaluation metrics for gas estimation. To solve this challenge, we search for several machine learning and deep learning methods:

Random forest, K Nearest Neighbors (KNN), Support Vector Machine (SVM), and two different evaluation metrics: Mean Average Percentage Error (MAPE) and accuracy rate. The performance for each regressor is discussed in Sect. 4.

Random Forest. Decision tree learning algorithm can build a regression model in a tree structure. It is prone to overfitting when a tree is very deep. So random forest comes out to minimize this error. A random forest [3] is a group of decision trees and aggregates their results to a final result. Based on the voting strategy, the random forest may produce a better result from assembled models rather than individuals. The random forest model has advantages of overcoming overfitting and can be more interpretable because it can explicitly output weight for each dimension of features.

K Nearest Neighbors (KNN). KNN [14] is a common used supervised learning algorithm. Suppose the distance of samples is defined by a similarity measurement, for example, Euclidean metric, Minkowski distance, Manhattan distance, etc. 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 weights of their real output value. KNN methods usually have better performance on datasets with smaller size.

SVM for Regression (SVR). Support Vector Machine (SVM) [2] is a widely used machine learning method. SVM constructs a margin separator which finds a hyperplane that has the maximum distance between features. The SVM method is first proposed for the classification and can be extended to regression, called Support Vector Regression (SVR), although traditional SVM is based on the linear separable assumption. By defining the inner product of features in terms of a kernel function, the SVM also suits non linearly separable problem. Intuitively, gas estimation is not a linearly separable problem. This inspires us to use the Gaussian Kernel function in our experiments.

LSTM. As traditional neural networks can not handle the context information of time series data, recurrent neural networks are proposed to solve this. The information of history data is preserved by introducing the time-variant hidden state for each network cell, and the relationship between inputs can be learned during gradient descent. The 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. It contains input, remember, and output gate, which gives LSTM network cells the ability to decide which values to through and abandon others. We can treat the input opcode sequence as a time series sequence, and each opcode can be represented by an embedding word vector. The LSTM aims to give a prediction of gas based on the new input opcode sequence.

3 Our Approach

Now we give the workflow of the proposed approach. There are mainly three steps shown in Fig. 1. For simplicity, we refer the transactions sent to loop functions as loop transactions. First, we collect input and receipt for existing

loop transactions. Then, we replay all loop transactions and extract their trace on local blockchain. Here *trace* means transaction executed opcode sequence on EVM. Last, we build a gas estimator model based on transactions trace using machine learning and deep learning algorithms. After gas estimator construction, for a new loop transaction, we can execute it on a local blockchain and get its transaction trace. By feeding this trace to the gas estimator model, the estimated gas cost is given.

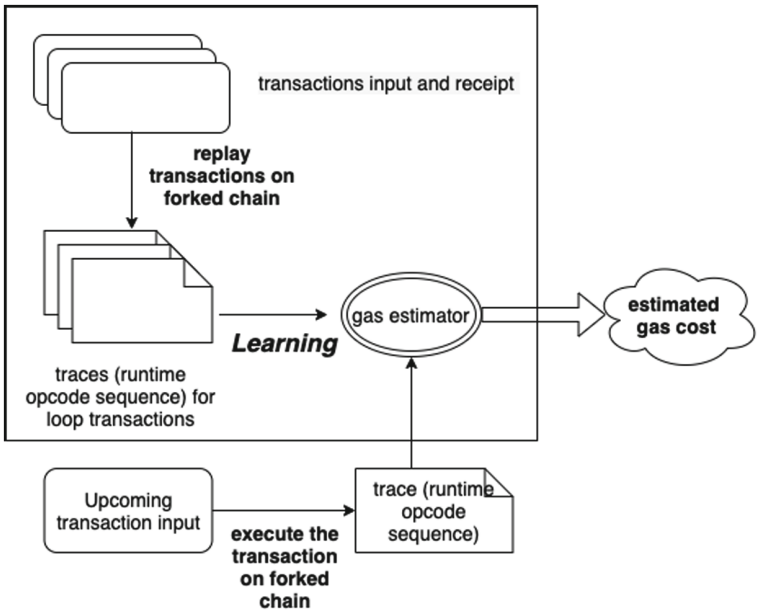

Fig. 1. Workflow of a trace-based approach

3.1 Loop Transactions 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 loop functions). Next, we gather existing transactions sent to this contract. By analyzing inputs for existing transactions, we collect transactions which sent to loop functions. Details for the selection and analyzing steps are shown as follows:

1. **Select loop functions:** We first use Slither [8] to get the control flow graph (CFG) for functions in contracts. Slither is a static analysis framework, which can convert Solidity contracts to an intermediate representation called SlithIR. SlithIR has a node type called "IF_ LOOP" indicating the start of a loop. And we traverse all functions CFGs to collect loop functions that have at least one "IF_ LOOP" node.

2. **Gather transactions to a contract:** Etherscan[3] shows all transactions hashes to a contract address. For this study, We crawled recently no more than 2000 transactions hash for considered contracts from Etherscan. Then we pull the transaction detailed information (e.g., input, transaction sender) from a full node on Ethereum mainnet by calling *web3.eth.getTransaction()* API. Especially, we deploy a full node based on QuikNode's node service.
3. **Analyze transactions sent to loop functions:** The input of a transaction contains the invoked function name and parameters. We use abiDecoder[4]to decode every transaction input to get the invoked function name. If the called function name is one of loop function names, we add this transaction information into our database.

3.2 Transaction Trace Generation

The transaction trace is the transaction execution opcodes sequence on EVM. A method to generate traces is calling the API *debug.traceTransaction()*[5] from a full node. However, it is slow to use this API to obtain the trace triggered by a transaction. Because for a given transaction hash, it needs to find the previous block that the transaction resides and then replay all preceding transactions before the transaction on the same block [6]. Also, a further time delay is caused by the Remote Procedure Calls communication from API. Chen et al. [6] proposed another way to record traces. They instrument a full Ethereum node and replay all transactions during synchronization. After synchronization is finished, the traces are automatically collected. They aim to collect traces for all transactions. But it is costly for us to replay some specific loop transactions using their method.

We propose a new way to get 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 $\#N_b - 1$ block to start a local testnet. Here we use **Ganache-cli**[6] and **Infura**[7] node service to implement this. Ganache-cli is part of the Truffle suite and the command-line version of Ganache. It can build a personal blockchain for development. Especially, 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 on mainnet. Infura is a node cluster that free developers from synchronizing and maintaining an Ethereum node. Our study hosts an archive node because it can respond to API requests for any historical blocks. As shown in Fig. 2, our local testnet share the chain starting from the genesis block to $\#N_b - 1$ block. This is to construct the correct the same state before the original transaction. Especially, we revised the Ethereumjs-VM to collect trace when replaying transactions. Ethereumjs-VM is the Ethereum

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

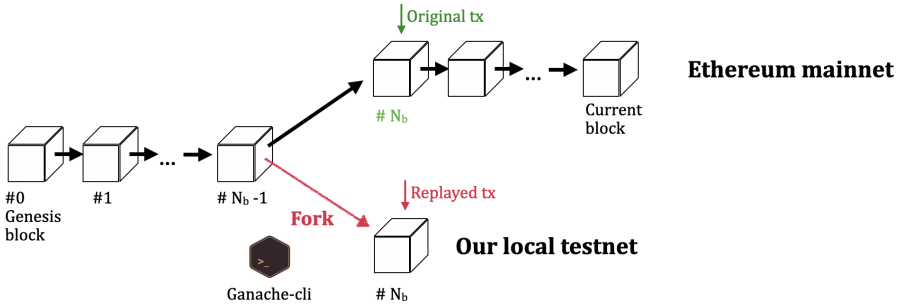

Fig. 2. Collect runtime trace by replaying a transaction

Virtual Machine (EVM) used in the Ganache blockchain. More concretely, EVM interpreter executes each opcode on **runStep** function, so we insert opcode recording code in this function. When EVM executes a transaction, the trace is automatically collected.

3.3 Build Gas Estimator Models

Trace for loop transactions are usually long. The maximal trace we collected contains 382,552 opcodes. Some works [10,15] using deep learning algorithms on malware detection based on input opcode sequence. In their practice, they only take the first L opcodes to meet the need for a deep learning network of the unified input length. As they observed, the larger the L is, the more memory and computation time is required to train the neural network. For gas estimation, we cannot simply follow this rule because each opcode contributes to the final predicted gas. Also, in our experiments, memory overflow error is raised due to the long sequence, even with batch size 1. It is hard to feed this long sequence into any existing learning models directly. So we propose two kinds of abstractions as features, i.e., opcodes frequency and dynamic opcodes sequence, shown in Fig. 3.

There are a total of 141 opcodes used on EVM. We checked go-ethereum[8] source code and divided them into three classes: constant cost opcodes, dynamic cost opcodes, and both constant and dynamic cost opcodes. EVM charges 117 opcodes and 10 opcodes in constant and dynamic gas costs, respectively. For example, **ADD** opcode costs 3 gas, and **EXP** gas cost can only be decided runtime. In addition to constant gas cost, a total of 14 opcodes also have dynamic gas, such as **SHA3**, which has fixed 30 gas cost and dynamic cost relating to memoryGasCost.

In Fig. 3, the frequency-based method extracts the frequency of all opcodes and feeds it to different learning models. For example, opcode 0x60 (i.e., PUSH1) occurred nine times in original trace (opcode sequence), so its frequency is 9. Here we consider three supervised machine learning models: random forest, K-nearest neighbors (KNN), and SVM for regression (SVR). Moreover, we have another

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

abstraction on the original long trace and propose a sequence-based method that only contains dynamic opcodes sequence but maintains the original opcodes order. Because only 24 opcodes have dynamic gas costs, so dynamic opcodes sequences shorten the original trace.

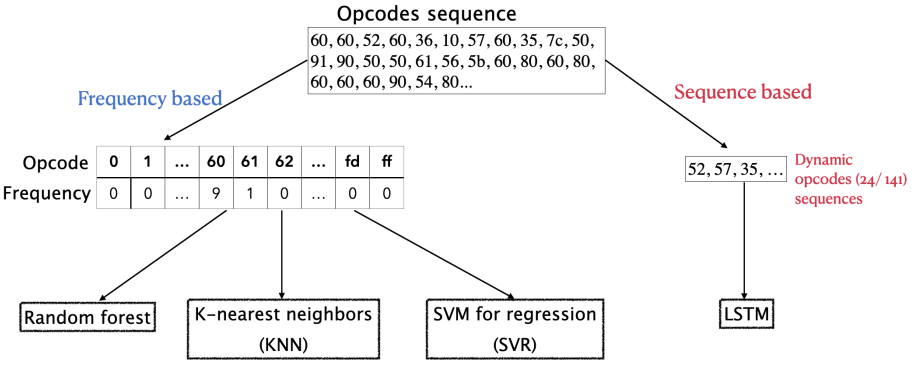

Fig. 3. Build gas cost models

After the gas estimator is learned, for a new transaction, we can first execute it on forked ganache blockchain and collect its trace, then get estimated gas by input its trace to the gas estimator.

4 Results

We use Smartbugs [7] contract dataset containing 47,398 unique contracts. As stated in Sect. 3, for each contract program, we employ Slither [8] 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 crawled the recently 2000 transaction records to 10, 855 contracts from Etherscan and analyzed transaction inputs. The results show that there are 706 contracts with transactions to loop functions. Up to 50 transactions are sent to each of 457 loop contracts, which amounts to 64.7% of all loop contracts. Besides, 64 loop contracts range in loop transactions from 500 to 2000, which occupy 9.1% of all loop contracts. Almost a quarter of contracts have loop functions, but users do not often send transactions to them. The reasons behind this might be: 1. smart contracts might contain loop-related vulnerabilities, such as unbounded loop [9], but there is no effective tool that can remedy them. 2. There is no practical tool to estimate the gas cost for loop functions.

As state in Sect. 3, we need to replay loop transactions on forked testnet and collect their transaction traces. In our experiments, the average transaction replay time is about 30 s. For a very complicated transaction, it took 3 min to

replay it. Considering time limits, we replayed recently *up to ten* transactions[9] for each loop contracts. Totally, we collect traces for 5718 transactions. The opcode length for these traces ranges from 43 to 382,552. The frequency-based method fixes 141 opcodes frequency as features. The sequence-based method maintains dynamic opcodes sequence as features and the maximal length is 14,267.

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

- **Mean Absolute Percentage Error (MAPE):** it expresses the error as a ratio defined in this 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 predicted gas and actual gas is divided by the actual gas, where the predicted gas is directly estimated by learned gas estimator. The smaller MAPE indicates the better prediction performance.
- **Prediction accuracy rate:** because the learned estimator may underestimate gas for some transactions, we compute this metric as different accuracy rates by adding additional gas on estimator provided gas. Here accuracy means that the predicted gas is higher than actual gas.

4.1 Frequency Based Method Performance

For frequency based method, besides collected 5718 transactions, we replayed transactions for four representative contracts, whose contracts hash are listed in Table 1. We first counted the opcodes frequency for each trace, then applied machine learning models (Random forest, KNN, SVR) on frequency vectors separately. The training set and testing set were randomly split into 70% and 30%, respectively. The training time is less then 2 s. The MAPE results are shown in Table 1. We have two observations:

- In general, gas estimation based on transactions to the same contract has a lower error rate than on transactions for different contracts. For example, if we use a random forest learning algorithm to estimate gas for transactions to contract 0x92240... and to combined four contracts transactions separately, the former MAPE is 0.78, and the latter is 1.99.
- In most cases, random forest and KNN can have a lower error rate than SVR. Consider contract 0x117cb..., the MAPE for random forest and KNN are 5.05 and 6.94, which is lower than 9.74 predicted by SVR.
- Recall that we replayed recent less than ten transactions for each loop contract and totally collected 5718 traces. The MAPE for these transactions is distinctly higher than that for combined four contract transactions.

Figure 4, Fig. 5, Fig. 6, Fig. 7, Fig. 8 and Fig. 9 list the prediction accuracy rate with incremented gas using random forest algorithm for six kinds of transactions on Table 1. Generally, the more gas is added to estimator provided gas, the more accuracy rate we can gain, where the accuracy rate means the percentage of the increased gas is higher than actual gas. For example, for transactions

[9] These ten transactions invoke the same loop function.

Table 1. MAPE results using Random forest, KNN and 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	1.99	1.94	110.90
706 contracts hash	5718	16.67	23.71	168.77

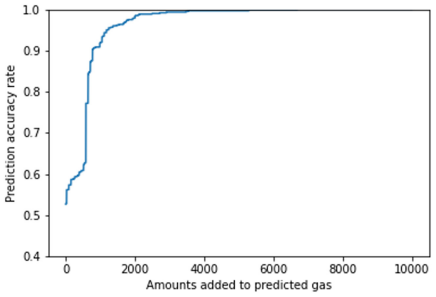

Fig. 4. Transactions to contract 0x611ce...

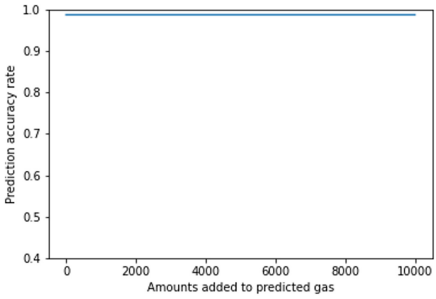

Fig. 5. Transactions to contract 0x92240...

to contract 0x117cb... , as seen in Fig. 6, if we add 2000 to predicted gas from gas estimator, the prediction accuracy rate can reach 70%, i.e., for 70% tested transactions, we can make sure that incremented gas is higher than actual gas. But if we add 6000 to predicted gas from gas estimator, the prediction accuracy rate can reach 82%,

4.2 Sequence Based Method Performance

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 contains dynamic opcodes) from each trace and fed these dynamic traces to LSTM models. The training set and testing set were randomly split into 70% and 30%. The training time is about three days. The MAPE for LSTM is over 800, which is far higher than MAPE for the random forest, KNN, and SVR.

Evaluate Our Method. Our methods are effective in estimating gas costs for loop transactions. The frequency-based method can have a distinctly lower

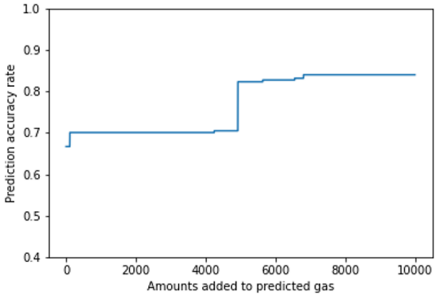

Fig. 6. Tranactions to contract 0x117cb...

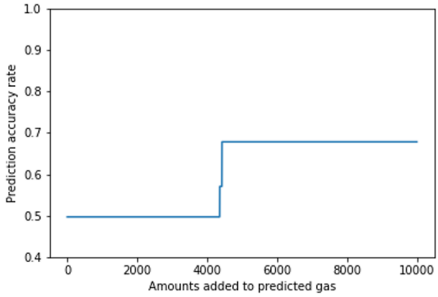

Fig. 7. Tranactions to contract 0x85b29...

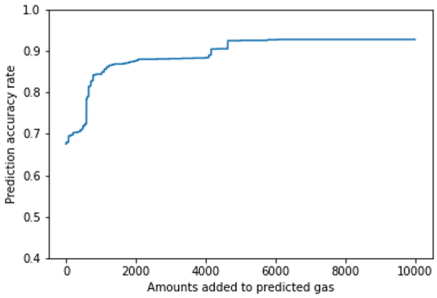

Fig. 8. Tranactions to four contracts 0x611..., 0x922..., 0x117..., 0x85b...

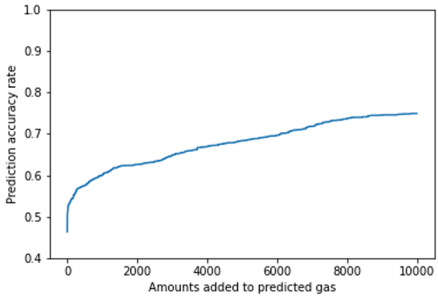

Fig. 9. Transactions to 706 contracts

prediction error rate (i.e., MAPE) than the sequence-based method. For the frequency-based method: 1. random forest and KNN have a better estimation rate than SVR. 2. prediction on transactions from the same contract is better than that from different contracts.

Limitation. 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 $\#N_b$ don't change the state of contract C. To mitigate this, we will try to analyze the relationships among transactions in the same block.

5 Related Work

Gas Estimation. Albert et al. constructed a gas analyzer named GASOL [1], which can over-approximate the gas consumption of functions. Also, Marescotti et al. [13] presented two methods to decide the exact worst-case gas consumption. Signer provided Visualgas [16], a tool to visualize how gas costs relate to different parts of the code. However, none of them compare the actual transaction gas cost on mainnet with their predicted one to prove the effectiveness of their methods.

Based on feedback-directed mutational fuzzing, Ma et al. designed GasFuzz to construct inputs which maximize the gas cost [12]. The Ethereum community also developed tools to help estimate gas costs. Solc statically predicts gas cost, but shows the infinite gas cost for loop functions. Web3[10] package can do gas estimation by executing the transaction directly in the EVM of the Ethereum node, but this only make sense when this transaction does not throw exceptions.

Gas Optimization and Vulnerability Detection. Chen et al. identified 7 gas costly patterns on Solidity code and developed GASPER [4] to locate 3 of them by analyzing bytecodes. They later listed 24 anti-patterns and implemented GasReducer [5] to detect and replace them with efficient code. They focus on optimize gas usage whereas we want to do gas estimation. Grech et al. [9] surveyed three gas-related vulnerabilities and detect them in bytecode level.

6 Conclusions and Future Work

In this work, we identify the importance of estimating gas costs for transactions to loop functions. We propose a trace-based approach to estimate the transactions gas. In the experiments, we extract opcodes frequency and dynamic opcodes sequence from transaction traces as two kinds of abstractions and feed them to different learning algorithms. The results show that our method is effective in estimating gas cost for loop functions. Especially, random forest and KNN have a better estimation rate than SVR and LSTM. In addition, we provide a dataset that contains 5718 traces for transactions to loop functions. The dataset suggests more research and calls for attention to estimate the gas cost for loop transactions.

In the future, we would like to apply our idea to estimate gas costs for other functions besides loops. Also, since the prediction accuracy rate is not so high as expected, we will consider other trace abstractions to improve our results.

Acknowledgments. This work was supported partly by JSPS KAKENHI Grant Numbers JP17H06099, JP18H04093 and 19K20269. In addition, this work was partially funded 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).

References

1. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: GASOL: gas analysis and optimization for ethereum smart contracts. In: 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020. Proceedings. Lecture Notes in Computer Science (2020, to appear)
2. Boser, B.E., et al.: A Training Algorithm for Optimal Margin Classifiers (2010). https://doi.org/10.1.1.103.1189

[10] https://web3js.readthedocs.io/en/v1.2.0/web3-eth.html.

3. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
4. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 442–446. IEEE (2017)
5. Chen, T., et al.: Towards saving money in using smart contracts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER), pp. 81–84. IEEE (2018)
6. Chen, T., et al.: Tokenscope: automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1503–1520 (2019)
7. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. arXiv preprint arXiv:1910.10601 (2019)
8. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15. IEEE (2019)
9. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Programm. Lang. **2**(OOPSLA), 1–27 (2018)
10. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: Android malware detection using deep learning on API method sequences. arXiv preprint arXiv:1712.08996 (2017)
11. Liu, C., Gao, J., Li, Y., Chen, Z.: Understanding out of gas exceptions on ethereum. In: Zheng, Z., Dai, H.-N., Tang, M., Chen, X. (eds.) BlockSys 2019. CCIS, vol. 1156, pp. 505–519. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-2777-7_41
12. Ma, F., et al.: Gasfuzz: generating high gas consumption inputs to avoid out-of-gas vulnerability. arXiv preprint arXiv:1910.02945 (2019)
13. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 450–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_33
14. Peterson, L.: K-nearest neighbor. Scholarpedia (2009). https://doi.org/10.4249/scholarpedia.1883
15. Santos, I., Brezo, F., Sanz, B., Laorden, C., Bringas, P.G.: Using opcode sequences in single-class learning to detect unknown malware. IET Inf. Secur. **5**(4), 220–227 (2011)
16. Signer, C.: Gas cost analysis for ethereum smart contracts. Master's thesis, ETH Zurich, Department of Computer Science (2018)
17. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper **151**(2014), 1–32 (2014)