Gas Estimation and Optimization for Smart Contracts on Ethereum

by

Chunmiao Li

Dissertation

submitted to the Department of Informatics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI
September 2021

Abstract

Ethereum 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. Any user can send a transaction for invoking functions 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. To prevent resource waste from infinite loops and make sure that contract programs can terminate at a certain point, the computation effort required to execute the EVM instructions is charged in the gas unit. For a transaction, the actual money cost that the sender needs to pay is the multiplication of gas units and the gas price. The gas price is specified by participating users before the transaction starts and remains the same for all opcodes in the EVM execution process.

Currently, two important issues are widely studied regarding gas. One is gas estimation, which aims to predict executional gas consumption before the transaction starts. Different approaches have been proposed for it, but it is still nontrivial in estimating gas costs for transactions to functions involving loops. On the one hand, the loop iteration times might rely on users' inputs and cannot be decided statically. On the other hand, dynamic methods often send transactions to the local testnet and observe the gas cost, whereas this gas usually differs from the actual transaction gas cost because the testnet cannot reflect the state change on the mainnet. This thesis proposes a gas estimation approach to dynamically estimate the gas based on the transaction trace. The insight is that the relationship between the historical transaction traces and their gas costs can be learned to estimate the gas for new transactions. Especially, three different abstractions of the original transaction trace are considered and fed to different machine learning models. The results show that the proposed approach is effective in gas estimation for transactions to loop functions and that a random forest can achieve the most accurate estimation.

The second issue is gas optimization. Users might be overcharged due

to inefficient programming patterns in smart contracts. Therefore, removing gas-inefficient patterns can reduce gas costs or transaction fees. However, we find two problems for gas optimization that previous work did not notice: one for storage usage and the other for array access. For storage usage optimization, we recognize that existing work only considered loop structures but not for all possible program behaviors. To mitigate this gap, we propose an approach to automatically optimize contract functions at the Solidity level so that each execution path of the contract functions can read or write the state variables with the least times. The experimental results show that (1) 44% of contracts have inefficient gas use conditions, and (2) our approach can, on average, save 1.03 USD and 1.16 USD for transactions with one function and one contract, respectively. For the array access optimization part, existing approaches mostly work on data of basic types (e.g., int, bool), and it is not clear how to minimize the gas costs for compound data structures, although compound data are mostly used in deployed smart contracts. To mitigate this condition, two novel and distinct gas-inefficient patterns concerning bound checks for the arrays are found in this thesis. Then, an approach is devised to replace the problematic array accesses in that two gas-inefficient patterns with assembly code. A tool GotBucks is developed to automatically optimize gas-costly array accesses in smart contracts. The evaluation shows that: (1) 33.9% contracts involving array accesses contain the first gas-inefficient pattern, and 34% include the second pattern; and (2) on average for each optimized contract, 3.17 USD and 0.27 USD can be saved for optimizing the first and second proposed patterns, respectively.

Acknowledgements

When I was a master student at Shanghai Jiao Tong University, I was lucky to be an intern in the Programming Research Lab of the National Institute of Informatics (NII). The academic atmosphere there impressed me a lot and professors were enthusiastic about researches, so I pursued a PhD in NII after finished my master study. Three years have passed and there comes the graduation.

First and foremost, I am especially grateful to Professor Zhenjiang Hu for his delicate supervision of my research. He continued supervising me even after joined Peking University. Prof. Hu patiently guided me to survey the related works and finally found the key research problem. He always gave me wise suggestions and strong support. Moreover, when I sprained my knees, he drove me to the hospital immediately and generously prepaid the medical expenses for me. I would not recover so fast that time without his kind help. I also want to thank Professor Okada Hitoshi, who agreed to be my nominal supervisor in the last two years of my PhD.

I must thank Dr. Cao Yang, an Assistant Professor at Kyoto University. He gave me a lot of useful advice for doing research and kindly modified my draft papers. He shared famous blockchain books and useful posts for me to extend my research idea. I must also thank Prof. Yu Yijun, a senior lecturer at the Open University, UK. He is passionate about researches. He provided much valuable feedback on my research.

Besides, I want to thank all lab members: Josh Ko, Zirun Zhu, Zhixuan Yang, Yongzhe Zhang, Liye Guo, and VanDang Tran for having a great research life

together. They shared many field expertise and provided precious advice on my study. Especially, the Chinese lab mates introduced me cheap but secure house and accompanied me for lunches and dinners on working days. I did not feel too lonely with their company. I also would like to thank all professors in the lab: Kato sensei, Sekiyama sensei, Takechi sensei, and Tsushima sensei. They introduced many interesting works in lab meetings and fostered my understanding of other research fields.

I would like to thank the members of my dissertation committee, Professor Kato, Professor Sekiyama, Professor Okada, and Professor Yoshioka for giving insightful comments to improve my work.

Thank my boyfriend Shijie Nie. During my doctoral studies, he was my best friend. There were so many days we discuss researches and write papers. I could not finish my study without his continuous encouragement.

Finally, I appreciate the deep love and support of my parents. They provided me tremendous spiritual and financial supports all time, which enabled me to focus on my study. I was so lucky to be their daughter. Hope they enjoy their life in good health.

Chunmiao Li National Institute of Informatics & Tokyo, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter includes four sections. First, a general background of this thesis is presented. Then, the motivations and objectives are discussed. Finally, the thesis organization and contributions are listed.

1.1 Background

As a decentralized shared ledger, blockchain has attracted much interest from academic and industrial areas since it was first proposed by Satoshi Nakamoto [79]. Blockchain records the ordered transactions from all participants. Currently, two types of blockchains are mainly considered: public and private (or semiprivate) blockchains. The former (such as Ethereum [104] or Cardano [15]) allows each participant to join the consensus creation process. In the latter type of blockchain (Hyperledger Fabric [17], R3 Corda [18], etc.), only a limited number of participants can achieve that process. The DApp Market Report [1] clearly demonstrates that almost half of all functioning DApps on the market live in the Ethereum network, which has become the most prevalent public blockchain.

[1] https://dapp.review/article/251/2020-Q1-Dapp-Market-Report-by-DappReview

Ethereum [104] builds a "world computer" that can host and execute programs. These programs are so-called smart contracts [104]. 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. As Figure 1.1 shows, 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.

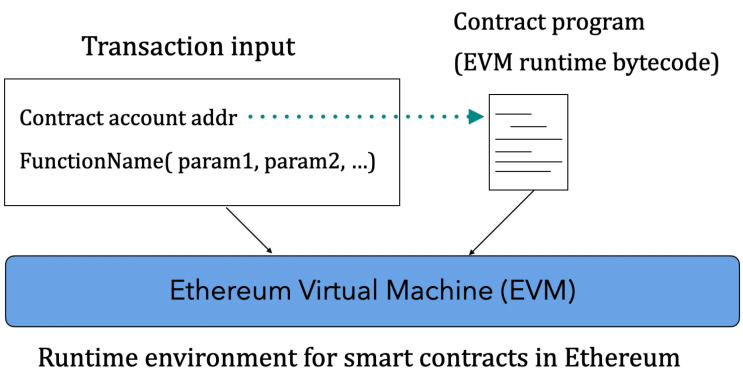

Figure 1.1 EVM runs contract bytecode according to transaction input data.

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. The EVM gas model [104] defines the gas cost for each opcode. The gas costs for some opcodes are listed in Table 1.1. In addition to function call- and contract creation-related opcodes, the Ethereum storage access opcodes, i.e., **SLOAD** and **SSTORE**, charge the highest gas cost [2] among all opcodes, and this is far higher than the costs for other opcodes (e.g., 3 gas for the **ADD** opcode) because Ethereum storage opcodes are used for

[2]Their gas costs are defined in EIP-2200

updating the Ethereum state and take a long time to process.

Table 1.1 The gas costs for some EVM opcodes.

Opcode	Gas cost	Description
ADD	3	Addition operation
MULMOD	8	Modulo multiplication operation
PUSH1	3	Place 1 byte item on the stack
SLOAD	800 [19]	Load a word from storage
SSTORE	20000	Save a word to storage (when the storage value is set to a non-zero value from zero)
SSTORE	5000	Save a word to storage (when the storage value's zeroness remains unchanged or is set to zero)
CREATE	32000	Create a new account with the associated code
CALL	700	Message-call into an account

When a user sends a transaction, the user needs to specify a gas limit attached to the transaction. 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. For example, consider the contract function in Listing 1.1. The gas requirements for executing an `f1` is related to the length of the transaction input `recipients`. The loop iteration times are determined by user input, and thus, the maximal gas cost for the loop is unbounded. If the length is too long, the gas cost required to execute the loop function might become so significant that it exceeds the block gas limit, and this transaction will not be included in any block. In other words, regardless of how much a user can afford, this transaction will always fail.

```
function f1(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));
  }
}
```

Listing 1.1 A function with unbounded gas cost

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. The gas cost studied in this thesis 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 1.1. The intrinsic gas is calculated directly according to [104], whereas the execution gas cost is complex and almost impossible to accurately compute.

$$G(tx) = G_{intrinsic}(tx) + G_{execution}(tx) - G_{refund}(tx) \tag{1.1}$$

1.2 Motivations and Objectives

Currently, two important issues are widely studied regarding gas: gas estimation and gas optimization. In the next two subsections, we state the motivation and our objectives for these two issues, separately.

1.2.1 Gas Estimation

Once exceptions occurred in EVM running, EVM will immediately return to the state before the current execution and consume all gas limit that the transaction sender provides. According to the survey in Liu et al. work [68], the economical losses due to exceptions amount to thousands of US dollars. There are many kinds of exceptions, such as out-of-gas, explicit revert, and illegal instructions. Especially, the out-of-gas exception accounts for over 90% of all exceptions on Ethereum and causes substantial financial losses[68]. Therefore, it is essential to avoid out-of-gas exceptions in EVM.

The root cause for the out-of-gas exception is that, for a transaction, the actual gas cost is greater than provided gas cost (i.e., gas limit). The most direct

way to prevent out-of-gas is to do gas estimation, i.e., given a transaction, we need to estimate its gas cost. However, 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 [29], 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 [44]. It is unknown which EVM client the version miner will adopt. In addition, the execution gas may depend on the state of the blockchain [21], which continues updating with new transactions. For example, it is difficult to statically estimate an accurate gas cost for the transactions to the function shown in listing 1.2 [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.

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

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

```
12            + uint32(fs.quality) * 100 + fs.pos); } }
```

Listing 1.2 The iteration times of loop is decided runtime.

Some studies have been devoted to gas estimations [71, 96, 74, 21]. For example, Solc [4] statically predicts the gas cost for all contract functions. Marescotti et al. [74] apply symbolic model checking methods to detect the worst-case gas cost. Albert et al. designed GASOL [21], a gas analyzer for over-approximating the gas consumption of a function.

We found that a quarter of all contracts have loop functions, and the gas costs for loops are higher than those 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). Static analysis cannot determine the number of iterations of any loops; hence, Gasol [21], 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.

Therefore, the first target of this thesis is to estimate gas costs for transactions to loop functions, so that transaction senders can assign gas limit with estimated gas before the transaction starts. There are two requirements for the estimated gas. First, the estimated gas should be less than the block gas limit, otherwise, the transaction will not be considered to be included in a block. Besides, the predicted gas costs should be slightly higher than actual gas costs.

1.2.2 Gas Optimization

Because the transaction fee is equal to the product of the actual amount of gas used for the execution of the transaction and gasPrice, more gas used means a higher transaction fee paid. Previous work found that users might

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

be overcharged due to inefficient programming patterns in smart contracts. Therefore, removing gas-inefficient patterns can reduce transaction fees.

Gas optimization for smart contracts has received increasing attention in the literature [33, 23, 22, 34, 36, 78]. The general approach is to detect *gas-inefficient patterns* in smart contracts on either low-level opcode streams (or assembly) or high-level Solidity source code, and then replace them with gas more efficient smart contracts. For the low-level optimization, Albert et al. [23] and Chen et al. [33] replaced gas costly opcode sequences with synthesized and manually summarized cheaper counterparts, respectively. For example, the sequence "*SWAP1/DUP2/SWAP1*"(9 gas) can be replaced by "*DUP1/SWAP2*"(6 gas). Additionally, the built-in optimiser[5] for Solidity compiler applied a group of predefined optimization steps, such as DeadCodeEliminator to the assembly. For high-level Solidity source code optimization, prior work [33, 22] listed several gas-costly patterns on high-level Solidity source code that can be replaced with gas more efficient ones. Chen et al.[33] suggested that the expensive operations accessing Ethereum storage in a loop can be lifted outside the loop. Albert et al. [22] extended that idea and transformed multiple accesses to the same storage data in this way: introduced a local variable s_0 to hold the storage data, then replaced all accesses to the storage data by accesses to s_0, and if s_0 was modified, assigned s_0 to the storage in the end.

However, we found that two aspects for gas optimization that previous work did not notice: one for storage usage and the other for array access. For storage usage optimization, we recognize that existing work only considered loop structures but not for all possible program behaviors. For the array access optimization part, we observe that none of the existing work considered the optimization for compound types, such as arrays. In other words, existing methods were designed for basic data types (e.g., int, bool), which cannot optimize compound data structures like arrays. Actually, arrays are mostly used in smart contracts involving loops. To verify this, we collected 10855 contracts containing loops deployed on Ethereum and found that 9759 (90%) of them used arrays in at least one function. Thus, it is worthwhile to minimize the gas costs for array access. Therefore, the second aim of this thesis is to

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

mitigate the two limitations of existing work.

1.3 Organization and Contributions

The remainder of the thesis is organized as follows. Chapter 3 - Chapter 5 describe the main contributions of this thesis, which include the approaches and implementations for gas estimation and optimization.

Chapter 2: We introduce previous work on gas estimation and optimization in this chapter. Besides, we also list vulnerability detection work in smart contracts on Ethereum. Since these vulnerabilities may affect the correctness of the smart contracts, they will still be necessary to adopt for the smart contracts generated by gas optimization.

Chapter 3 (Gas estimation): 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. Especially, 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.

Chapter 4 (Gas optimization): We identify various inefficient storage gas use conditions. Besides, we propose an approach to automatically optimize contracts at the Solidity level so that each execution path of the contract functions can read or write the state variables (or Ethereum storage) with the least times. We implement our approach in three algorithms. The experimental results show that (1) 44% of contracts have inefficient gas use conditions; (2) our approach can, on average, save users 1.03 USD and 1.16 USD for transactions with one function and one contract, respectively.

Chapter 5 (Gas optimization): We find two novel and distinct gas-inefficient array accesses in smart contracts and propose the *first* approach to replace them with assembly. We build a tool GotBucks to automatically optimize the gas costs

in smart contracts, and applied to 10,304 deployed contracts involving array accesses. The results show that: (1) 33.9% contracts involving array accesses contain the first gas-inefficient pattern, and 34% include the second pattern. (2) We can, on average for each optimized contract, save 3.17 USD and 0.27 USD for optimizing the first and second patterns proposed, respectively. Last, we establish an evaluation method to compare the gas difference for two contracts on the Solidity level.

Chapter 6: We conclude the thesis by reviewing the contributions and elaborating the future work.

Chapter 2

Related Work

The gas-relevant studies on Ethereum are depicted in Figure 2.1. In this chapter, we first recall previous work on gas estimation and gas optimization. Then, we present other work surrounding gas on Ethereum, such as gas cost mechanism, gas price prediction, and gas-oriented vulnerabilities. Last, we discuss the gas concepts in other blockchains.

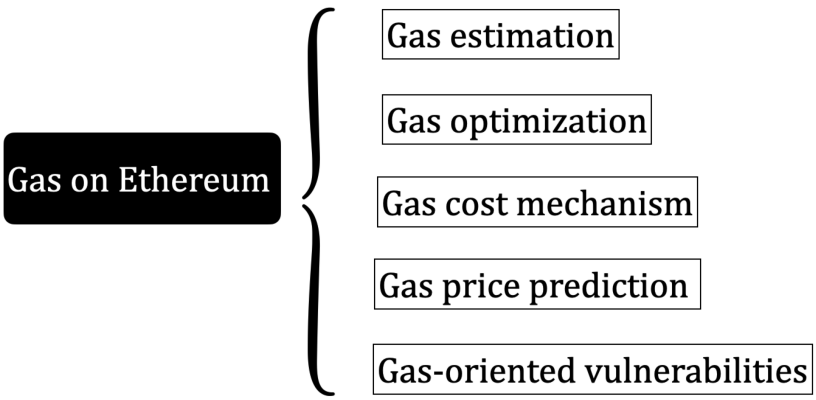

Figure 2.1 Gas related studies

2.1 Gas Estimation

Albert et al. designed GASOL [21], a gas analyzer for overapproximating the gas consumption of a function. In addition, Marescotti et al. [74] proposed two approaches for determining the exact worst-case gas consumption scenario. Signer implemented a tool named Visualgas[96] to visualize how gas costs are related to different parts of the code. However, the actual gas costs of transactions 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 gas costs [71] and could reduce the underestimation ratio to 13.86%.

The Ethereum community has also developed a set of tools for estimating gas costs. For example, Solc[1] can statically predict gas costs but fails for any function with loops. Remix[2] provides a debugger for listing the gas cost of a transaction execution trace. Additionally, the Web3[3] package can perform gas estimation by executing the examined transaction directly in the EVM of the Ethereum node, but this only makes sense when this transaction does not throw any exceptions. Paterno et al. presented a tool called gas exactimation[8], 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. However, this tool is currently still under review. Eth-gas-reporter[7] can show the gas cost after a transaction has finished.

[1] https://github.com/ethereum/solidity
[2] https://github.com/ethereum/remix
[3] https://web3js.readthedocs.io/en/v1.2.0/web3-eth.html

2.2 Gas Optimization

2.2.1 Optimization at the Program Level

There are two main kinds of studies regarding gas optimization on smart contracts: inefficient gas detection and automated gas optimization.

First, some works proposed gas-inefficient patterns that users should avoid when writing contracts. Chen et al. proposed seven gas cost patterns for the Solidity code and designed GASPER [34] to locate three of these patterns by analyzing their bytecodes. In their later work [36], they listed and detected 24 anti-patterns in smart contracts. In addition, Mudit Gupta [49] provided several tricks in his blogs, such as not needing to initialize variables with default values. Kaden Zipfel [108] gave four patterns for reducing gas consumption. Different from these works, we aim to automatically find contract-specific gas-inefficient patterns from the view of saving storage costs.

Second, many automated gas optimization tools have been designed. The Solidity compiler solc [20] was built with two optimizer modules: the "old" optimizer that operates at the opcode level and the "new" optimizer that operates on Yul IR code. Chen et al. [36] summarized 24 anti-patterns in the bytecodes of contracts and performed bytecode-to-bytecode optimization. Nagele et al. presented the super-optimization idea [78] for optimizing contracts. They encoded a sequence of instructions within a basic block as SMT formulas and found cheaper bytecodes using a constraint solver. Albert et al. [23] also employed a super-optimization approach to optimize contract instruction sequences. They summarized a functional stack specification from a block and synthesized a new optimized block with a minimal gas cost; this block has the same functional stack specification as that of the extracted version. In addition to optimizing bytecode sequences, a few studies have refactored Solidity programs to save gas. Albert et al. [22] replaced multiple accesses to the same position in the storage by using gas-efficient memory accesses. Their optimization approach is based on the number of SLOAD and SSTORE instructions. They only referred to one pattern (i.e., the state variable in the

loop should be replaced by a local variable) and optimized it in the source code. Since they did not open source codes, we cannot compare our work with their methods. Tamara [30] identified 19 optimization strategies for software engineering that can be applied to Solidity contracts to reduce gas costs. He also developed a prototype that could detect 9 of these rules and automatically optimize 6 of them. In contrast, our automatic gas optimization target storage usage and array accesses. Since the patterns are orthogonal, we cannot compare our optimization results with the other optimizations directly, but in principle results are additive.

2.2.2 Optimization at the Blockchain Level

Hukkinen [56] fixed two particular sources of inefficiency, i.e., replacing a contract function with off-blockchain communication and editing the fund withdrawal mechanism so that fewer blockchain transactions need to be created. Li et al. [67] presented GRuB, which dynamically replicates data between off-chain cloud storage and the blockchain. The GRuB system can achieve low gas costs with changing workloads. Zhang et al. [107] proposed a novel authenticated data structure, called the GEM^2-tree, which can be efficiently maintained by the blockchain.

2.3 Gas Cost Mechanism

Yang et al. [106] confirmed that gas costs for some opcodes are not correctly aligned with actual computational cost, and in transaction costs, the effects of I/O are poorly captured by the current gas cost model. Pierro et al. [90] discussed the factors that influence Ethereum transaction fees.

2.4 Gas Price Prediction

Go Ethereum client Geth [4] considered the minimal gas price of previous blocks to suggest a proper gas price for the current transaction. EthGasStation [5] introduced a gas price oracle to estimate gas price using a Poisson regression model of the previous 10,000 blocks. Sam M. Werner et al. [103] proposed a mechanism combining a deep-learning-based price estimation model and an algorithm that parameterized by an urgency value to forecast the gas price.

The actual transaction fee that the sender needs to pay is the multiplication of gas costs and the gas price. The gas price is specified by participating users before the transaction starts and remains the same for all opcodes in the EVM execution process. Therefore, the studies regarding gas price prediction and the gas estimation techniques proposed in this thesis can be considered together, so that the transaction fee can be better estimated.

2.5 Security Protection in Smart Contracts

2.5.1 Gas-oriented Vulnerabilities

Grech et al. [46] studied three kinds of gas-focused vulnerabilities: unbounded mass operations, non-isolated external calls, and integer overflows; they analyzed the entire blockchain and found that 4.1%, 0.12%, and 1.2% of all contracts are susceptible to those vulnerabilities, respectively. Chen et al. [35] observed that if the costs of operations are fixed, exploitable underpriced operations might always exist. To solve this issue, they designed a novel gas cost mechanism that adjusts the EVM operating costs in terms of their execution times to avoid DoS attacks. David Prechtel et al. [91] showed that 96.3 thousand (57%) contracts deployed on a live Ethereum network might be affected by gasless send issues. Peng et al. [85] introduced an instruction-level symbolic stack

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

[5] https://ethgasstation.info/

simulation method to detect gas vulnerabilities and synthesis attacks. Liu et al. [68] collected exception transactions on Ethereum and found that out-of-gas exceptions are the most common and destructive type of exception.

Although such gas-related vulnerabilities are related, they are not the focus of this thesis. With our results, it is anticipated that such attacks are less likely to succeed. We leave this for further study.

2.5.2 Other Vulnerabilities

The security of smart contract is a serious issue in Ethereum and have received attention from industry and research fields. The existing work on securing smart contracts are mainly divided into two parts: one is for smart contract and the other is for EVM (i.e., the execution environment of smart contracts), as shown in Table 2.1.

The previous studies on smart contracts are further classified into three kinds at different times durations: the time when writing smart contracts, time after writing smart contracts, and time when running contracts. We briefly discuss each of them as follows.

- When writing contracts: To help users write safe contracts, Anastasia et al. [75] designed *VeriSolid* framework to generate solidity code from the verified transition-system based models. The library openzeppelin-solidity [14] contained some safe arithmetic operations that thwart integer underflow and overflow attacks. Truffle [6] provided a set of useful suites, such as a personal blockchain Ganache to assist developers to easily deploy and test their contracts.
- After writing contracts: Oyente [70] used symbolic execution to detect several security bugs. TeEther [63] applied static analysis to explore control flow paths that reach attackable instructions. KEVM [53] formally defined the EVM in K semantics so that a contract can be verified using the KEVM virtual machine.

[6] https://www.trufflesuite.com/

- When running contracts: Li et al. [64] presented a tool named Solythesis, which instruments solidity contracts with a user-specified invariant, to reject all invariant violated transactions. Shaun Azzopardi et al. [27] developed present a tool *ContractLarva* to achieve runtime verification of Ethereum Smart Contracts.

Moreover, some work [76, 87, 26, 40] summarized common vulnerabilities, such as re-entrancy bugs. Since these vulnerabilities may affect the correctness of the smart contracts, they will still be necessary to adopt for the smart contracts generated by our approach for gas optimization. We leave that as a direction of future study.

2.6 Blockchain and Gas

The gas mechanism on different blockchain platforms are listed in Table 2.2[7]. Among seven popular platforms, we observe that gas is only applied in two blockchains: Ethereum and NEO. Different from the EVM computational costs in Ethereum, the gas in the NEO blockchain refers to a GAS cryptocurrency. Because the script language for Bitcoin doesn't allow loops, there is no need to apply gas to avoid denial of service (causing by infinite loop).

[7]The data is collected on May 2020.

Table 2.1 The overview of approaches for securing smart contracts.

Smart contracts level	When writing contracts	Auxiliary materials (Still Solidity/Serpent) [16, 39]
		New programming language describe contracts [89, 95, 92, 13, 45, 61]
		Intermediate representation [75, 43]
		(Higher) regulated SC standards [5, 6]
		Contract development tools (Employ predefined strucutures/libraries/ frameworks) [14, 31, 12]
		Program synthesis [98]
		Domain specialized templates based [73]
		Borrow secure patterns from other PLs [83]
	After writing contracts	(Automated) Testing [12, 3, 69, 105, 81]
		Release new patched compiler version [57]
		Visualization [11]
		Automated Analysis tools [70, 91, 82, 77, 69, 101, 42, 100, 99, 59]
		Auditing (like outsourcing security check) [10, 4, 9]
		Run Bounty programs [2, 1]
		Formal verification on contracts [54, 25, 24, 102, 38]
	Running contracts	Online detection [48]
		Runtime Validation [27, 97, 93, 64]
EVM level	K framework [72], new semantics on EVM [47], protocol change [86], Others [55]	

Table 2.2 Gas mechanism on different blockchain platforms.

	Blockchain platforms	Gas mechanism
Permissioned	Hyperledger Fabric	No (DoS is easy to be prevented)
	R3 Corda	No
Permissionless	**Bitcoin**	No (scripts don't allow loops)
	Cardano	No (plan to implement gas in the future)
	EOS	No
	NEO	Yes
	Ethereum	Yes

Chapter 3

Trace-based Dynamic Gas Estimation of Loops

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 chapter, we propose a gas estimation approach based on the transaction trace to dynamically estimate the gas for the loop functions. The following sections are organized as follows. Section 3.1 provides the challenges we meet and solutions. Section 3.3 presents the workflow of our trace-based learning method. Before that, Section 3.2 introduces the learning models used in our experiments. Section 3.4 describes the experimental results and limitations. We give some concluding remarks regarding this research in Section 3.5.

3.1 Challenges and Solutions

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 several 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 the challenge (1), we use an Ethereum-js virtual machine [1] 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 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.

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

3.2 Learning Models

To the best of our knowledge, there are no previous studies applying learning algorithms to a gas estimation. In this thesis, we define a gas estimation as a work for 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 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 3.4.

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 [32] 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 overfitting and can be more interpretable because it can explicitly output weight for each dimension of the features.

K-Nearest Neighbor (KNN) A KNN [88] 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 better performance on datasets with a smaller size.

SVM for regression (SVR) An SVM [28] 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 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.

3.3 Our Approach

Now we will describe the workflow of the proposed approach. There are mainly three steps, as shown in Figure 5.6. 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.

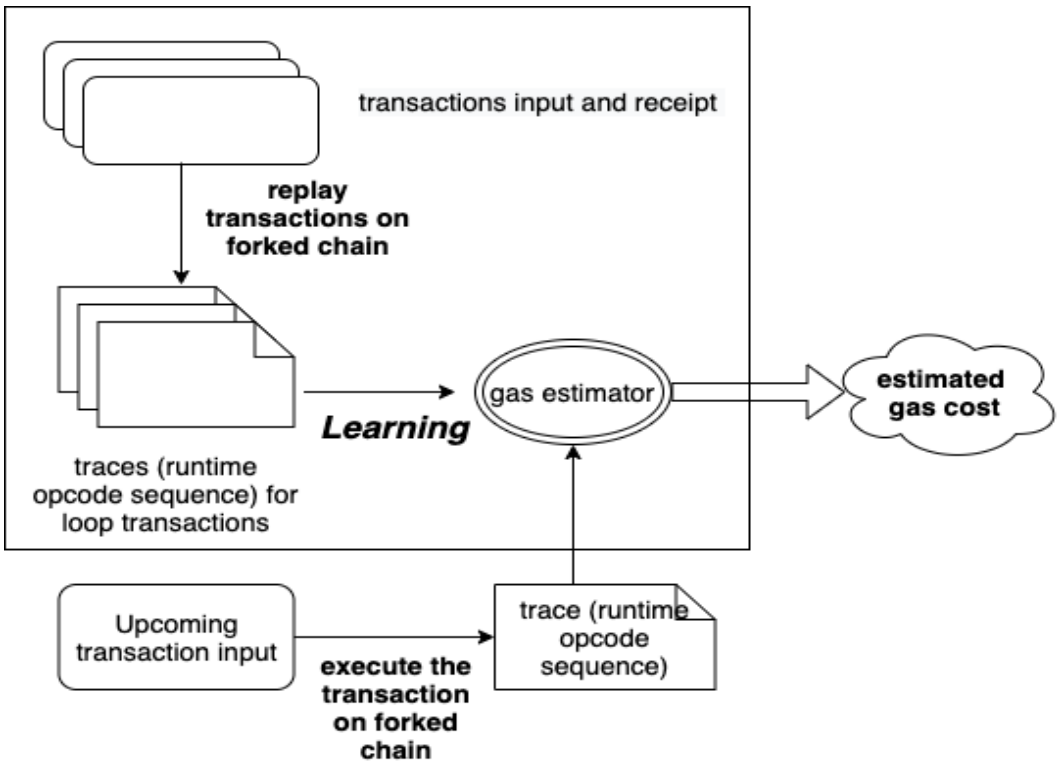

Figure 3.1 Workflow of trace-based approach

3.3.1 Loop Transaction Collection

A loop transaction is a 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 for selection and analysis steps are as follows:

1. **Select loop functions:** We first use Slither [42] 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[2] shows all transaction hashes sent to a contract address. For this study, we searched no more than 2000 recent transaction hashes for considered contracts 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[3] 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.

3.3.2 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()* [4] 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 [37]. In addition, a further time delay is caused by the remote procedure calls from the API. Chen et al. [37] 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

[2]https://etherscan.io/

[3]https://github.com/ConsenSys/abi-decoder

[4]https://github.com/ethereum/go-ethereum/wiki/Management-APIs

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 Figure 3.2. 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** [5] and a **Infura** [6] 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 Figure 3.2, 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.

3.3.3 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 [94, 60] 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

[5] https://github.com/trufflesuite/ganache-cli

[6] https://infura.io/

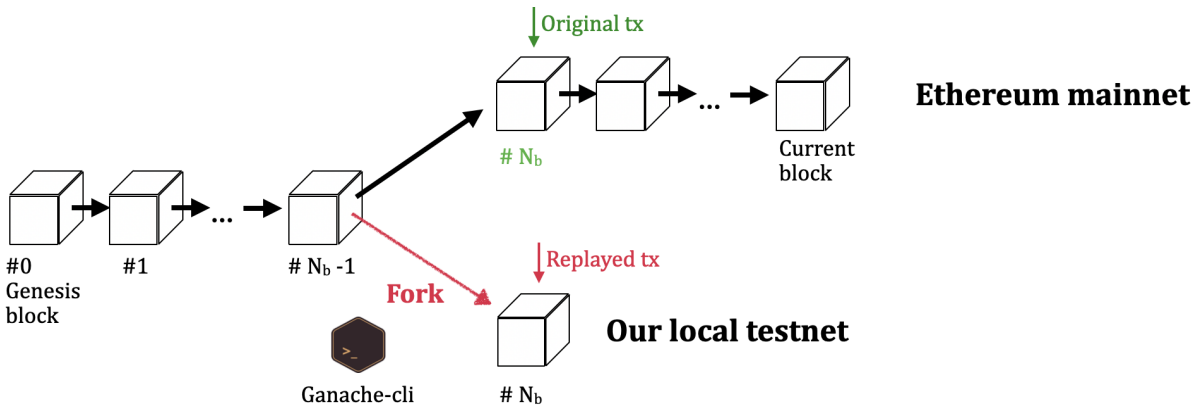

Figure 3.2 Collect runtime trace by replaying a transaction

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 [65], we extracted the opcode frequency and dynamic opcode sequence from the transaction trace. The frequency-based method extracts the frequency of 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 our later study [66], 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 Figure 3.7, 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.

Opcode Frequency

There are 141 opcodes used on an EVM. In Figure 3.7, 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.

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 Figure 3.7.

The loop function vector is constructed in four steps. First, we use Slither [42] to obtain the control flow graph (CFG) for a loop function. Second, we modify the Exas algorithm [80] to extract features of two types of patterns (n_path_pattern and node_pattern) from the control flow graph of a function.

- 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 in Figure 3.6, a feature of 3_path_pattern is "Node Type: BEGIN LOOP 6 $\rightarrow$ Node Type: IF LOOP 9 $\rightarrow$ Node Type: EXPRESSION 10".
- The node_pattern associates a node with p incoming and q outgoing edges. For example, node 9 in Figure 3.6 has two incoming edges from nodes 6 and 13, and two outgoing edges to nodes 7 and 10. Thus, the feature for node 9 is "Node Type: IF_LOOP 9 $\rightarrow$ 2 $\rightarrow$ 2".

We partly modified the vector computing algorithm in [80] to extract features

for a loop function, as algorithm 1 shown in Figure 3.3. 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 in Figure 3.4). Fourth, for a contract function, its vector is computed as the occurrence counts of its features among all features (algorithm 3 in Figure 3.5).

Let us consider the function in listing 1.2 to understand the construction of a loop function vector. The parts of the control flow graph for this function are shown in Figure 4.3. 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 Figure 4.3 are shown in Table 3.1. We collected 1124 features from all loop functions using algorithm 2. Thus, the *loop function vector* is a 1124-dimensional vector. As shown in Figure 3.7, 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).

Dynamic Opcodes Sequence

We checked the go-ethereum[7] source code and divided it into three classes: constant cost opcodes, dynamic cost opcodes, and both constant and dynamic cost opcodes, as shown in Table 3.2. 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

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

```
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
```

Figure 3.3 Algorithm1: get_function_feature(adapted from [80]).

Table 3.1 The patterns and features extracted from Figure 4.3

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	...

```
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
```

Figure 3.4 Algorithm2: get_all_features.

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 Figure 3.7, 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.

3.4 Experiments and Results

We use the Smartbugs[41] contract dataset containing 47,398 unique contracts. As stated in Section 3.3, for each contract program, we employ Slither[42] 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%

```
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
```

Figure 3.5 Algorithm3: get_function_vector.

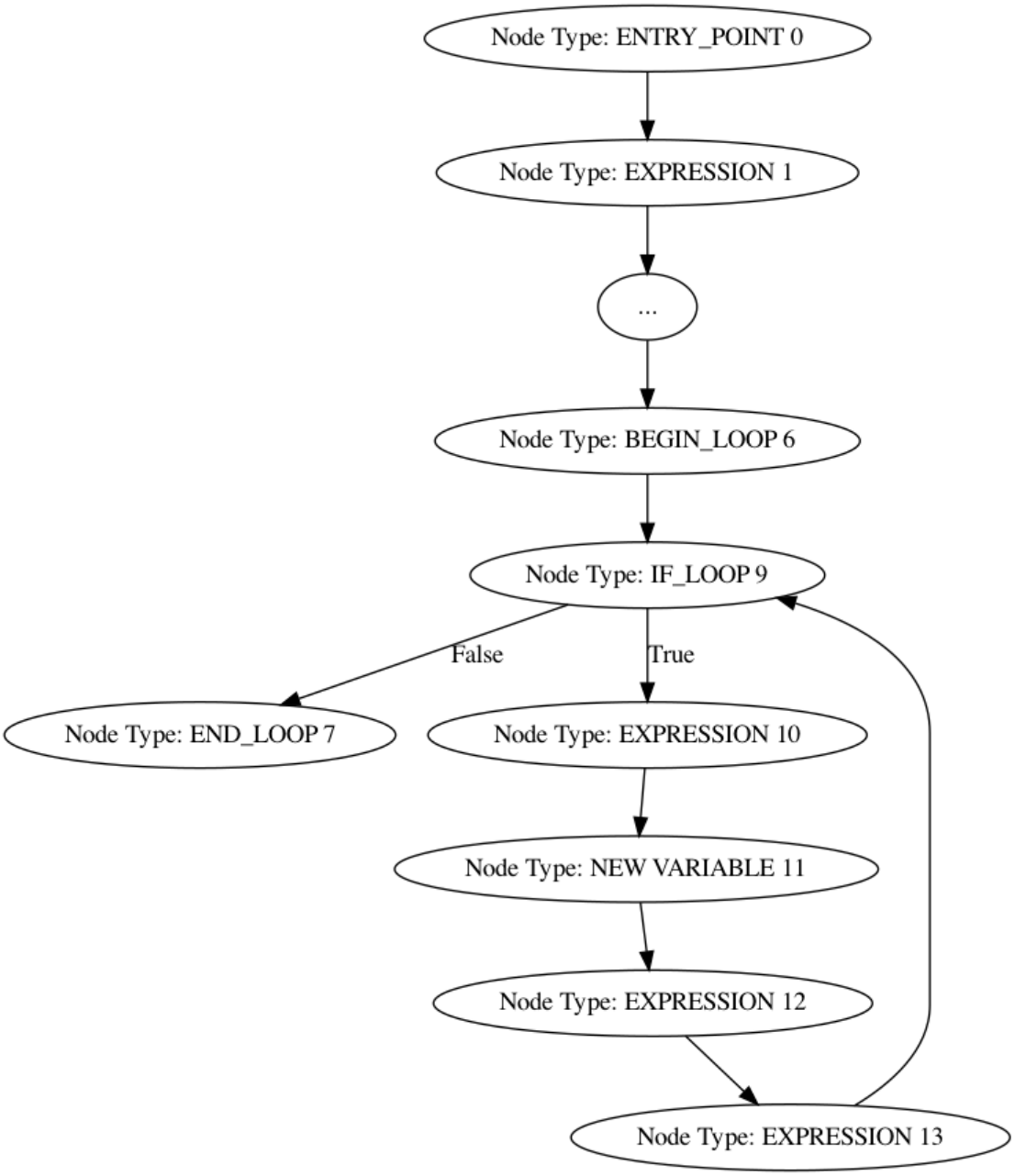

Figure 3.6 Control flow graph for function in listing 1.2

Opcodes num	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%

Table 3.2 Opcodes and their gas cost

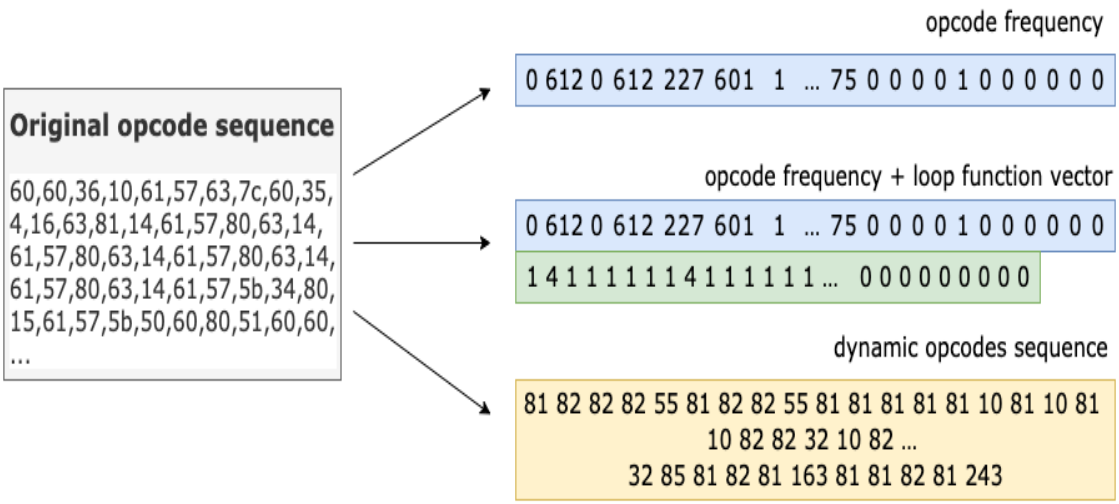

Figure 3.7 Three types of features for a transaction

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 [46], 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 3.3, 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[8] 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:

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

- **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.
- **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.

3.4.1 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 3.3. 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 3.3. We have three observations:

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

Table 3.3 MAPE results using a random forest, a KNN, and an SVR

Contract hash	Loop trans number	MAPE		
		Random forest	KNN	SVR
0x611ce69529...	2000	1.40	1.49	17.46
0x9224016462...	1238	0.78	0.59	0.96
0x117cb292e97...	790	5.05	6.94	9.74
0x85b2949cea65a...	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

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. Figure 3.9 depicts the MAPE results using three models in different numbers of loop transactions. We apply the same inspection as our previous study [65], i.e., the MAPEs for a random forest are generally lower than that for a KNN and an SVR.

Fig 3.8 list the prediction accuracy rate with an incremented gas using a random forest algorithm for the six types of transactions listed in Table 3.3. 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 3.8, 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 we add 6000 to the predicted gas from the gas estimator, the

prediction accuracy rate can reach 82%.

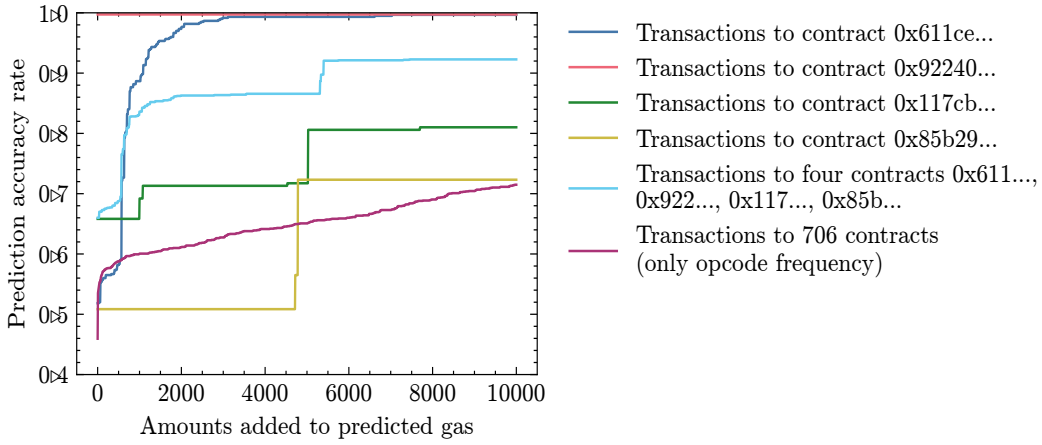

Figure 3.8 Prediction accuracy rate for transactions to different contracts

3.4.2 Combination of Opcode Frequency and Loop Function Vector Based Method Performance

As with the frequency-based method only, we chose 5718 transactions 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 [58] 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 3.4, we observe the following:

- For the same number of PCA components, a random forest model provides the smallest MAPE results.
- 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,

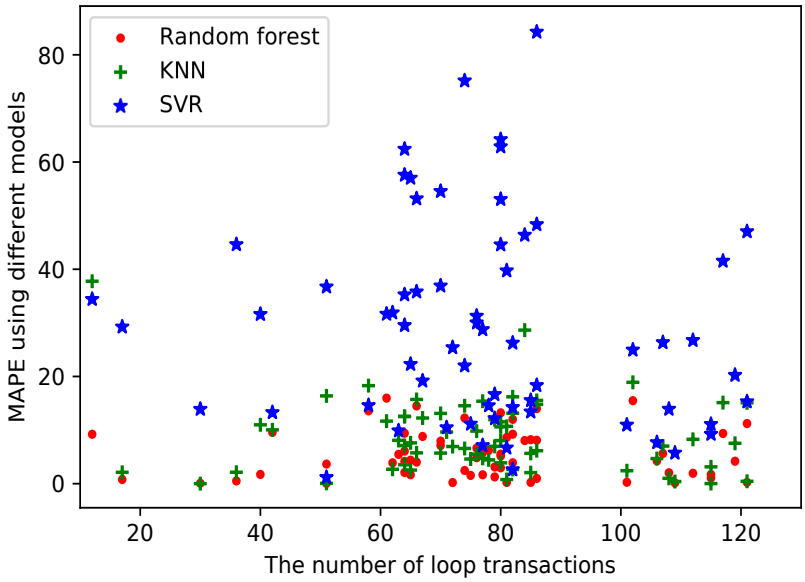

Figure 3.9 MAPE for different numbers of loop transactions

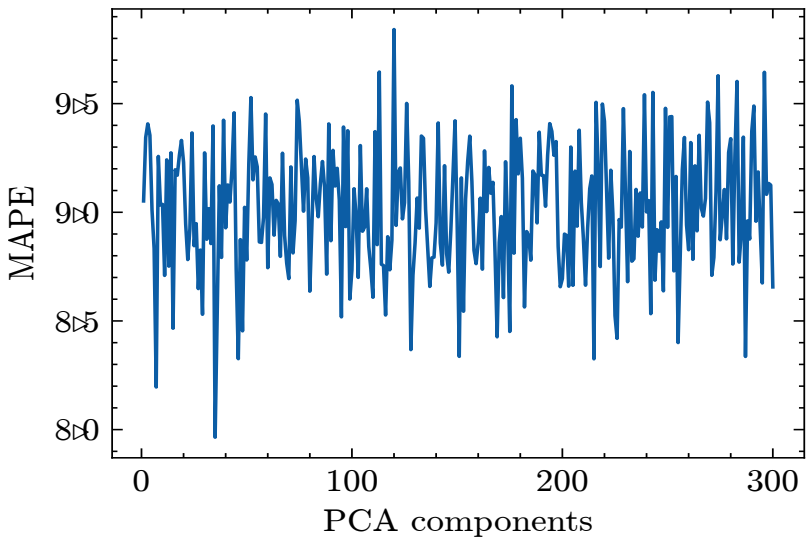

Figure 3.10 MAPE for different numbers of PCA components

Table 3.4 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

the MAPE for a random forest is 9.95. However, the subsequent numbers in the same column are all lower than 9.95.

Fig 3.10 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.

3.4.3 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.

3.4.4 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 has covered most types of contracts. So our method applies for gas estimation of more transactions to other new contracts. Besides, we can get a better prediction accuracy rate with more transaction traces.

3.4.5 Limitation

- As shown in Fig 3.2, 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.
- The gas relating to 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[104]: 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 the 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, 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 on the overall transaction gas cost.

3.5 Summary

In this chapter, 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 effective in estimating the gas cost for loop functions. In particular, a 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.

Chapter 4

Gas Optimization by Removing Inefficient Storage Usage

A transaction fee is equal to the multiplication of the actual amount of gas used for the execution of the transaction and gasPrice. The more gas indicates a higher transaction fee. Previous studies [33, 49, 23] found that contracts might contain some gas-inefficient programming patterns, which cause users to be overcharged. So it is necessary to optimize contracts to replace them with gas-efficient patterns.

In this chapter, we find novel inefficient storage gas use conditions that have not been realized before, and describe them in Section 4.1. Especially, we show a function suffering inefficient gas usage, and optimize the function using the existing approach and our approach. Then, we intuitively explain our idea using a running example in Section 4.2 and recall use-def relation in Section 4.3. The overall workflow of our approach is described in Section 4.4. Section 4.5 evaluates our method and analyzes our experimental results. We clarify some limitations and future work in Section 4.6. Finally, we conclude this chapter in Section 4.7.

4.1 Inefficient Storage Gas Usage

Chen et al.[33] and Albert et al. [22] optimized storage gas cost. They suggested that the operations required to access storage in a loop can be moved outside the loop. Because local and global variables in contracts are stored in stack and storage, respectively, concretely, for a global variable s, they introduced a local variable s_0 to hold the value of s before the loop, replaced all accesses to s by accesses to s_0, and if s_0 was modified, assigned s_0 to s after the loop. Because accessing s_0 requires much less gas than operating s, the modified contract is more gas-efficient.

We also aim to reduce storage-related gas because accessing storage is far more expensive than accessing stack [104]. However, as mentioned before, Chen et al. focused on storage gas optimization only for loop structures and did not consider all possible program behaviors. For example, a global variable might be read and written somewhere besides loop and Chen et al.'s approach can not be directly applied to contract programs with multiple branches and exits because the optimization needs to satisfy two requirements: (1) the local variables should be introduced only before the common start of different program execution paths that access global variables for many times. (2) compared with the original program execution path, the optimized one should access the storage with less or equal times.

To make the optimization target clear, we optimize the gas-inefficient contract function (Listing 1) in Figure 4.1 using existing approach [33, 22] and ours. The global variable s is stored in Ethereum storage, so the optimization aim is to reduce the access times of s to decrease storage gas costs. The Listing 2 in Figure 4.1 shows the optimized contract function **f** using existing method [33, 22], which introduces $s0$ to hold the value of s, replaces all occurrences of s by $s0$, and assigns $s0$ to s before **f** exits. The optimized **f** by our method is illustrated as Listing 3 in Figure 4.1. We insert $s0 = s$ in line 5 and keep s in line 15 unchanged but replace all other s by $s0$. Each version of function **f** in Listing 1-3 has three execution paths with No. 1-3, which are depicted in Table 4.1. For example, the original contract (Listing 1) has a path (No. 1) 3-4-5-6-15 where the numbers indicated the line numbers of statements, and this path is feasible

when $x > 12$(line 3) and $s > 2$(line 4). The access times of the global variable s is 3 (shown in parenthesis below this path in Table 4.1) from lines 4-6. Note that the global variable s is stored in Ethereum storage, so for path 3-4-5-6-15, we need to access storage three times. The corresponding paths in Listing 2 and Listing 3 are 3-4-5-6-7-16-17 and 3-4-5-6-7-8-17, separately. The access times of s for them are 2 and 1, which means that both the existing approach and ours reduce the storage gas costs when two determine statements (line 3-4) succeed. However, compared with the path 3-12-13-15 (feasible when $x <= 12$) in the original function, the corresponding optimized path 3-4-13-14-16 in Listing 2 accesses s twice but our approach still only access s once as the original function path, which means that the existing method induce extra storage gas costs but we don't. Compared with the original program execution path, the optimized function using our method always access the storage with less or equal times. So our approach is more appropriate than the existing one.

Table 4.1 The execution paths of three different program behaviors for Listing 1-3 in Figure 4.1. A number in a parenthesis indicates the accessing times of global variable s for the path. Compared with original program execution path, the optimized one by our approach always access s with less or equal times.

No.	Listing 1 Origin contract	Listing 2 Optimized contract by existing method [33, 22]	Listing 3 Optimized contract by our approach
1	3-4-5-6-15 (3)	3-4-5-6-7-16-17 (2)	3-4-5-6-7-8-17 (1)
2	3-12-13-15 (1)	3-4-13-14-16 (2)	3-4-14-15-17 (1)
3	3-4-9-15 (2)	3-4-5-10-16-17 (2)	3-4-5-6-11-17 (1)

The optimization of smart contract should not introduce extra gas and existing method [33, 22] can not satisfy this, especially for contracts with

Listing 1: The gas-inefficient contract function.

```
int s;
function f(int x, int y){
 if(x > 12){
   if(s > 2) {
      y = s;
      x = s * y;
   }
   else
      x = s * x - 1;
 }
 else {
   y = x * g(y);
   s = y + x;
 }
 return x + y;
}
```

Listing 2: The optimized contract function by existing method.

```
1  int s;
2  function f(int x, int y){
3   int s0 = s;
4   if(x > 12){
5     if(s0 > 2) {
6        y = s0;
7        x = s0 * y;
8     }
9     else
10       x = s0 * x - 1;
11  }
12  else {
13    y = x * g(y);
14    s0 = y + x;
15  }
16  s = s0;
17  return x + y;
18 }
```

Listing 3: The optimized contract function using our method.

```
1  int s;
2  function f(int x, int y){
3   int s0;
4   if(x > 12){
5     s0 = s;
6     if(s0 > 2) {
7        y = s0;
8        x = s0 * y;
9     }
10    else
11       x = s0 * x - 1;
12  }
13  else {
14    y = x * g(y);
15    s = y + x;
16  }
17  return x + y;
18 }
```

Figure 4.1 Listing 1 shows the original function. Listing 2 and Listing 3 list the optimized function using existing and our approaches.

multiple branches and exits, so we need a deep analysis to promise that the chosen optimization approach can definitively reduce storage access times but ensure that no extra gas is added.

4.2 A Running Example

In this section, we use a motivating example to explain our optimization results and perform some experiments to evaluate the performance of our approach.

Figure 4.2 shows an under-optimized contract with three global variables (*s*, *t* and *p*) and two functions (*f1*, *f2*). The line numbers are shown to the left of the frame. Because *f2* only reads variables s and p once each, we mainly consider possible optimizations for *f1*. The control flow graph of *f1* is depicted in Figure 4.3. The label of each node represents the line number, and the notation next to each node shows the read or write times for a global variable. If there is more than one statement in the same line, we label them by "line number:statement". For example, the notation "r(t_1):1, w(t):1" for node 9 stands for "the statement in line 9 reads the local definition of *t* once and writes once to *t* (separately)". Here, the local definition of *t* refers to the fact that the value of *t* used here is not the original value of *t* when entering *f1*. Because global variables are stored in Ethereum storage that is shared by all nodes, it is expensive to access global variables.

However, as shown in Figure 4.3, we observe that multiple accesses to each state variable exist in all paths starting from the entry node to the exit node. Therefore, any program execution path potentially accesses each state variable many times. This means that each transaction invoking *f1* might charge a large amount of gas.

The automatically optimized contract using our approach is shown in Figure 4.4. We modify the original *f1* so that the number of times each state variable is accessed (read or write) is at most 2 for each program running path. We introduce three local variables *mirror_s*, *mirror_t* and *mirror_p* to be the caches for global variables *s*, *t* and *p*, respectively. The optimized *f1* has the same program

Listing 4: An under-optimized contract.

```
contract Example{
  uint256 public s;
  uint256 public t;
  uint256 public p;

  function f1(uint256 x, uint256 y) {
    y = p + x;
    if (x > 10) {
       for (t = 0; t < 20; t++){
            y += x * s;
            p = y;
       }
    } else {
       x = s - p + y;
       for (x = 5; x < 20; x++)
              s += y * t;
       p = s;
    }
    f2 ();// function f2 reads s, p
    return y+x;
  }

  function f2 () returns (uint256){
    return s+p;
  }
}
```

Figure 4.2 An under-optimized contract.

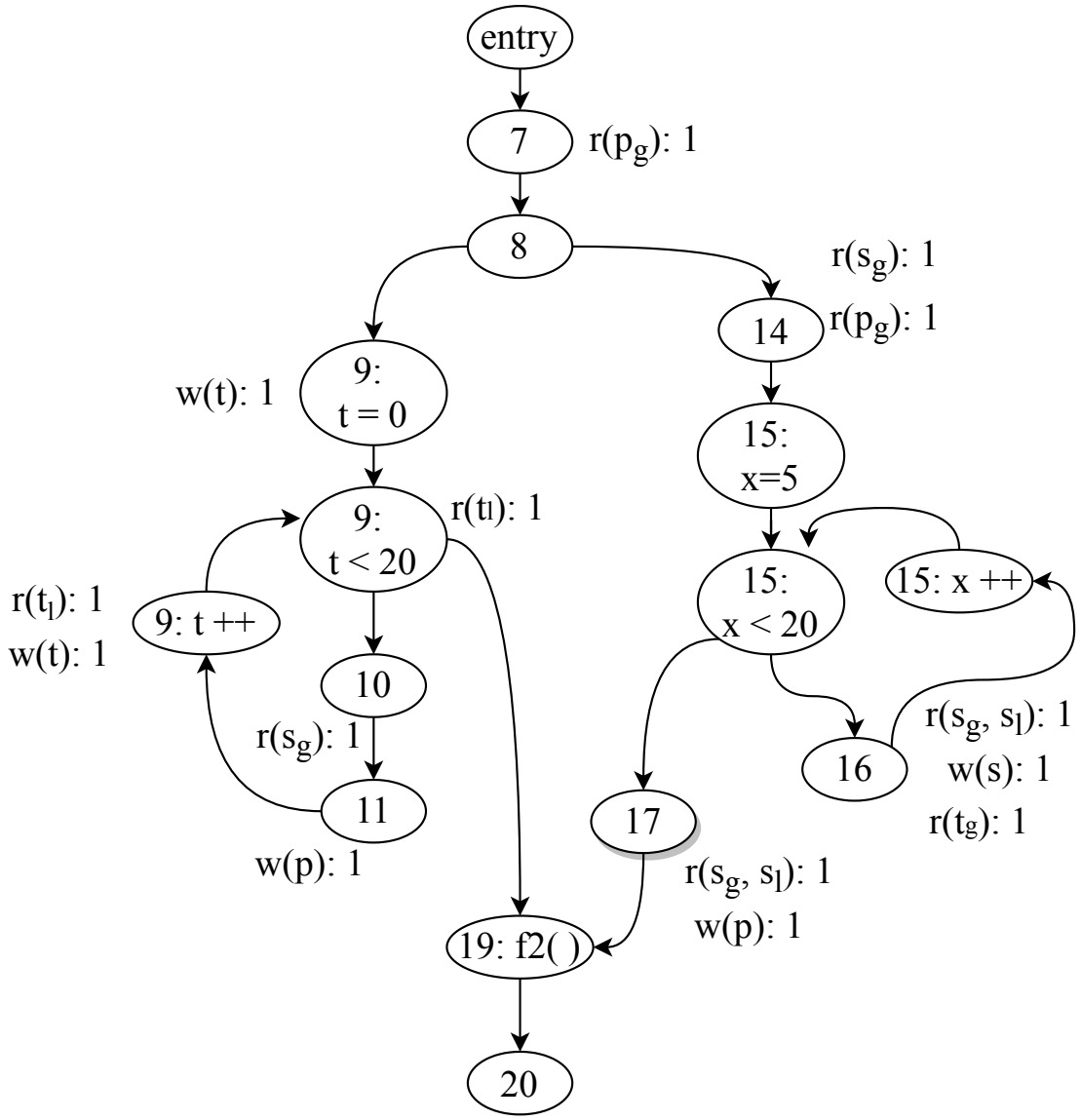

Figure 4.3 Control flow graph (solid edges connected) for function f1 in Figure 4.2. The numbers in nodes indicate the line numbers of statements.

semantics as the original *f1*. For example, the loop in lines 9-12 of Figure 4.2 is modified to lines 13-18 in Figure 4.4. In the latter optimized *f1*, the access times of *t* is reduced to 1 in line 18, which contains an assignment from *mirror_t* to *t*. The control flow graph of the optimized *f1* is shown in Figure 4.5. The label of each node represents the line number, and the notation next to each node shows the number of reads or write operations for a state variable. If there is more than one statement in the same line, we label them by "line number" suffixed by the number of statements in the same line. For example, 22_3 represents the third statement "x++" in line 22. Note that one major difference between the optimized *f1* in Figure 4.4 and the original one in Figure 4.2 is that each state variable is read once and written once at most in all program paths.

The Solidity compiler solc [20] has a built-in optimizer that can be turned on manually. To verify the effectiveness of our optimization process, we first compile these two contracts separately with solc optimization on and off. We collect four kinds of contract bytecodes and deploy them in Rinkeby [1] testnet. To make it clear, they are:

- the original contract without solc optimization,[2]
- the original contract with solc optimization, [3]
- the optimized contract with solc optimization, [4]
- and the optimized contract without solc optimization. [5]

The gas costs for the transactions required to deploy these bytecodes are listed in Table 4.2. With the solc optimizer, regardless of whether the original contract or optimized contract is used, the deployment costs are reduced by over 30000 gas. We confirm that the solc optimizer is useful for optimizing contract deployment costs. If we turn off the solc optimizer, the optimized contract using our approach can also decrease the cost by 636 gas. However, if we turn on the optimizer, our optimization cost increases by 2400 gas over that

[1] https://rinkeby.etherscan.io/

[2] This contract address is 0xf13265a5cb5519a162bc81dfbba36f62c4c2a120.

[3] This contract address is 0x5d9f036f132d9f5a142ff26f4959dcdc3ec84d2a.

[4] This contract address is 0x77c35a92a4e4b4eb499e7dea7d459bcf26267065.

[5] This contract address is 0xdb21ddd19249ab54cf3069c0eecbce2664221ed6.

Listing 5: The optimized contract for Listing 4.

```
contract Example{
 uint256 public s;
 uint256 public t;
 uint256 public p;

 function f1(uint256 x, uint256 y) {
  uint mirror_s = s;
  uint mirror_t;
  uint mirror_p = p;
  y = mirror_p + x;

  if (x > 10) {
   for (mirror_t = 0; mirror_t < 20;
    mirror_t++){
    y += x * mirror_s;
    mirror_p = y;
   }
   t = mirror_t;
  } else {
   mirror_t = t;
   x = mirror_s - mirror_p + y;
   for (x = 5; x < 20; x++)
    mirror_s += y * mirror_t;
   mirror_p = mirror_s;
   s = mirror_s;
  }
  p = mirror_p;
  f2 (); // function f2 reads s, p
  return y+x;
 }

 function f2 () returns (uint256){
  return s+p;
 }
}
```

Figure 4.4 The optimized contract for Figure 4.2.

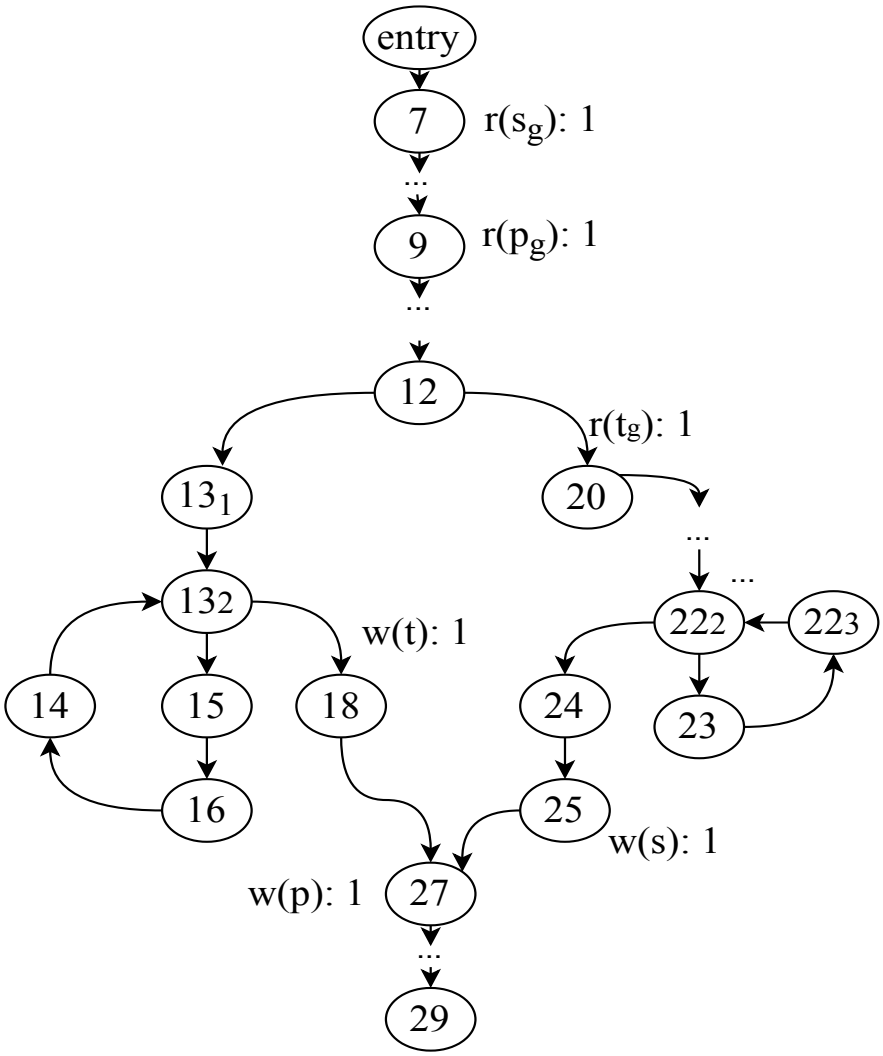

Figure 4.5 Control flow graph (solid edges connected) for function f1 in Figure 4.4. The numbers in nodes indicate the line numbers of statements.

Table 4.2 The gas costs for contract deployment in Figure 4.2 and Figure 4.4.

	Solc no optimizer	Solc with optimizer	Diff
Original contract	170569	136801	-33768
Optimized contract	169933	139201	-30732
Diff	- 636	+2400	

of the original contract. This might be due to the initialization of three local variables (lines 7-9) and additional assignments from these local variables to state variables (line 25) in Figure 4.4, which needs additional disk space to store the code. However, because the transaction to deploy the contract is called only once, in Figure 4.6 we see that one function call on our optimized contract can save at least 30000 gas, and this counteracts the increase in the deployment cost, so our optimization is still suitable.

To observe the gas costs differences for transactions invoking $f1$ in origin and optimized function $f1$, we call the four deployed contracts with *f1(20,8)* and *f1(8,10)*. These two function calls can pass two different branches, i.e., lines 13-18 and lines 20-25 in Figure 4.4 or lines 9-12 and lines 14-17 in Figure 4.2. The gas costs for these transactions are depicted in Figure 4.6. The pink bar and orange bar represent the contracts that are and are not optimized by solc, respectively. For example, for transactions invoking *f(20,8)*, the gas costs are 145841 and 146395 for the original contract with solc optimization turned on and off, respectively. In our optimized contracts, regardless of whether the solc optimizer is used, the gas cost is approximately 66000, which is less than half of that required for the same function call in the original contract. We conclude that for four kinds of function calls, solc optimization can only reduce gas

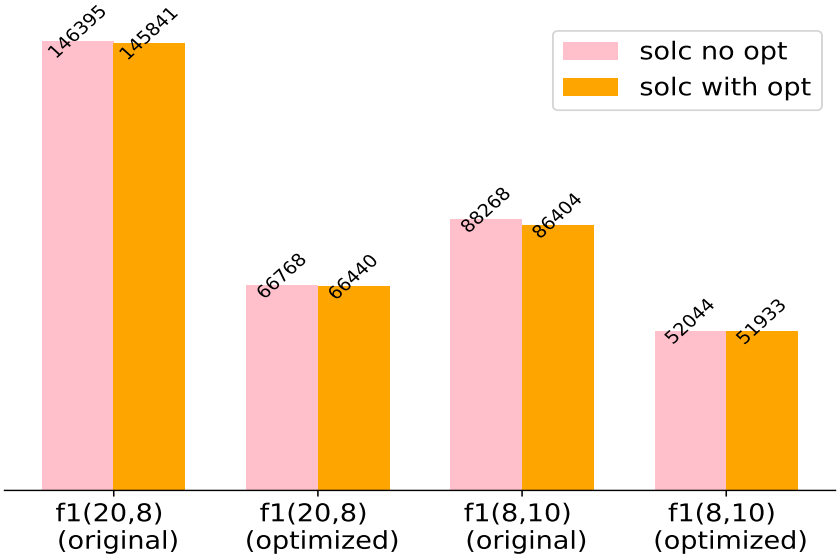

Figure 4.6 The gas costs of transactions to invoke f1 in Figure 4.2 and Figure 4.4.

costs by a small fraction. In contrast, our approach can significantly decrease the amount of gas required with or without compiler optimization. The main reason for this is that our approach reduces the number of accesses to storage during the execution of the function. For instance, lines 9-12 in Figure 4.2 are modified to lines 13-18 in Figure 4.4; the former access global variable t for many times, while the latter is only accessed once.

4.3 Use-def Relation

Before elaborating our approach, we now recall the idea of the use-def relation in dataflow analysis, which is applied in our approach.

The use-def relation refers to each use of a variable v in a statement s and a list of definitions of v that reach s. This relation can be computed by a reaching definition analysis [62] of the program. Concretely, a definition *def(v)* of a variable v is an assignment to v. We call *def(v)* can reach q when program point

p reaches another point q if there is no intervening *def(v)* in a path from p to q. For example, in the program in Figure 4.7, definition d1 can reach u2 and u3 but not u1 because d2 assigns a new value to v. Thus, the use-def relation for this program is $[(u1, d2), (u2, d1), (u3, d1)]$. Considering the CFG in Figure 4.3, node 16 reads the global definition of the global variable t because there is no intervening assignment to t in any path from the entry node to node 16. However, node 9:(t++) reads the local definition of t since node 9 assigns a new definition to t.

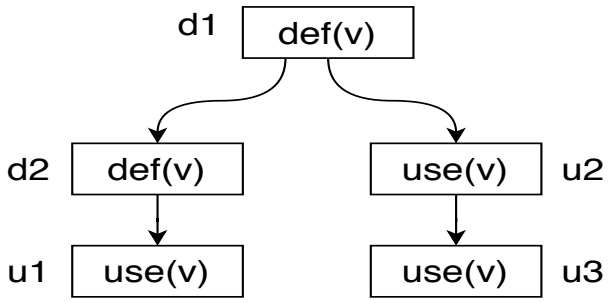

Figure 4.7 u1 uses d2, u2 and u3 use d1.

4.4 Our Approach

In this section, we present the workflow of our approach for achieving gas optimization on smart contracts. We first define our optimization target as follows.

Definition 1. *Given a contract C, we build an optimized contract C′ such that, for each function f in C, C′ has a corresponding function f′, and for each execution path p in f, f′ has a semantically equivalent execution path p′, where the storage access times in p′ are less than or equal to those in p.*

The overall workflow is depicted in Figure 5.6. Briefly, our optimization procedure consists of two steps. The first step is collecting candidate positions in

contract function for subsequent refactoring/optimization. Concretely, given an under-optimized function, we construct its control flow graph (CFG hereafter) and abstract syntax tree (AST hereafter). By applying Algorithm 1 in Figure 4.9 on the CFG, we collect three kinds of node lists for a global variable *t*: *list1*: contains nodes where *t* is replaced by a mirror variable *mirror_t*; *list2* and *list3* include the nodes that are followed by "*mirror_t* = *t*" and "*t* = *mirror_t*", respectively. The second step is to refactor the original function in terms of three node lists. We map CFG nodes to text position ranges on the function. Then, according to the text positions of nodes in *list1, list2, and list3*, we refactor their corresponding AST nodes and then un-parse the modified AST to produce an optimized function. We explain these two steps in the next two subsections.

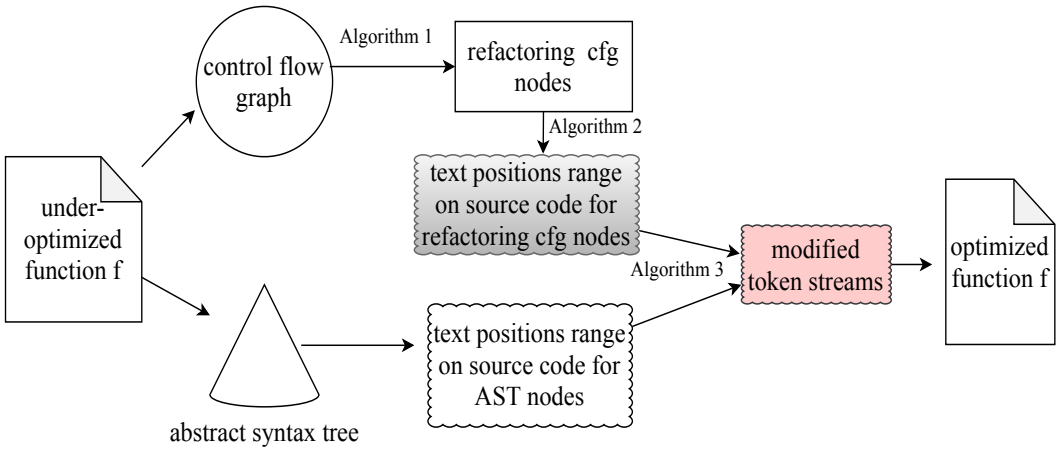

Figure 4.8 Overall workflow of our method.

4.4.1 Collecting CFG Nodes for Refactoring

Suppose a contract function accesses a global variable *s* for many times, we design Algorithm 1 (Figure 4.9) to collect three CFG node lists that indicates the refactoring places.

Now, we explain the details of Algorithm 1. The inputs are the CFG of a

INPUT: The control flow graph *CFG* of a contract function,
a global variable *s* in the function.
OUTPUT: Three kinds of nodes lists:
list1 contains the nodes where *s* is replaced by *s*0; *list2* and *list3*
include the nodes that are followed by "*s*0 = *s*" and "*s* = *s*0".

```
1:  for node N ∈ the CFG do
2:      if N calls f(), and f() only reads s then
3:          if the out-degree of this node != 0 then
4:              create a temporary node T
5:              for outer edge (N → P) from N do
6:                  remove (N → P)
7:                  create a new edge (T → P)
8:  for each connected sub-graph g of the CFG do
9:      deleteTag= True
10:     while deleteTag == True do
11:         deleteTag = False
12:         for path p from the start node to the leaf node in g do
13:             acMode = get access mode(p, s)
14:             if acMode ∈ [(0r,0w),(0r,1w), (1r,0w),(1r,1w)] then
15:                 remove all nodes on p from g
16:                 deleteTag = True
17:     for node N' ∈ g do
18:         if N' reads/writes s then
19:             add the node to list1
20:     g1, g2 = g, g
21:     deleteTag= True
22:     while deleteTag == True do
23:         deleteTag= False
24:         for path p' from the start node to the leaf node in g1 do
25:             if p' does not read the global definition of s then
26:                 remove all nodes on p' from g1
27:                 deleteTag= True
28:     add all entry nodes in g1 to list2
29:     deleteTag= True
30:     while deleteTag == True do
31:         deleteTag= False
32:         for path p' from the start node to the leaf node in g2 do
33:             if p' does not write s then
34:                 remove all nodes on p' from g2
35:                 deleteTag= True
36:     add all leaf nodes in g2 to list3
37: return list1, list2, list3
```

Figure 4.9 Algorithm 1: Collect CFG nodes for refactoring.

contract function and a state variable s [6]. The output is three kinds of node lists that are to be refactored. Lines 1-7 dispose of the function call. If the called function reads s, then we see the function call node N as an exit node for the function. As in Figure 4.10, we create a temporary node T and replace all outer edges like (N $\rightarrow$ P) with (T $\rightarrow$ P); this may split the CFG into different subgraphs. Starting from line 9, we analyze each connected subgraph. The access mode of a path from the start node to the leaf node is defined as the number of reads and write operations of s in the path. For example, in path *7→8→14→15(x=5)→...20* of Figure 4.3, for the state variable s, the access mode is (17r, 15w). Lines 9-16 aim to remove all nodes from the subgraph for the paths in which the access mode belongs to [(0r,0w),(0r,1w), (1r,0w),(1r,1w)] because this kind of path accesses s at most once and no optimization exists for this. Then, lines 17-19 put all remaining nodes with the read or write s operations into *list1*. We will modify the occurrence of s in these nodes using a local mirror variable s_0 in Section 4.4.2. We initialize two graph g_1 and g_2 using remained sub-graph g in line 20.

Next, we determine the nodes that are followed by "$s0 = s$" in lines 21-27. Note that in lines 25-27, we remove all nodes along the path without reading the global definition of s. Here, the global definition refers to the definition of s when entering the function. For example, in Figure 4.3, for the state variable t, path *7→8→9:t=0→...→19→20* does not read the global definition of t, so we remove all nodes in this path. Last, according to lines 33-35, we remove all nodes along the path without writing the state variable so that we write s in all paths of the remaining graph, and we put all leaf nodes into *list3*, which are followed by "$s = s0$".

We apply Algorithm 1 (in Figure 4.9) to collect CFG nodes to be refactored for function *f1* in Figure 4.2. The control flow graph for *f1* is shown in Figure 4.3. Figure 4.11(a) shows the subgraph g_1 in line 28 of Algorithm 1, where the start node 14 colored in pink is put into *list2*. Figure 4.11(b) depicts the subgraph g_3 in line 36. Node 11, colored in blue, is put into *list3*. So, for the state variable t, the returned CFG nodes *list1, list2 and list3* are [9:t=0, 9:t¡20, 9:t++,16], [14], [11],

[6] Although we consider a global variable in this algorithm, the steps are the same for other global variables.

respectively.

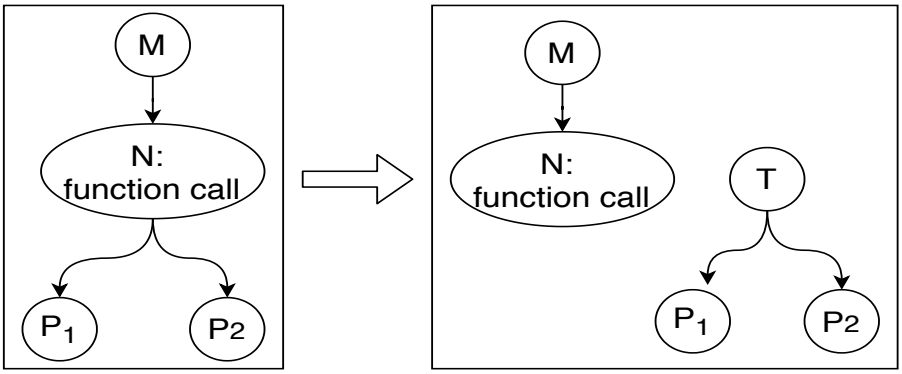

Figure 4.10 Create a temporary node T and replace all outer edges like (N → P) with (T → P).

4.4.2 Refactoring a Solidity Function

In this section, we refactor contract function according to the three CFG node lists collected in the last subsection. Usually, the refactoring process is conducted on an AST. Therefore, we need to first map the node in CFG to a syntax node in AST of the function. Because there is no direct mapping from CFG to AST, we use the text position range of the CFG node as a bridge. In Figure 4.13, the text position range for CFG node 11 is [353,357], relating to line 11 "p = y" in the source code, and this statement is parsed into three syntax nodes, "p", "=", and "y", in the AST. Then, we modify AST and unparse this modified AST to produce an optimized function.

Before illustrating our refactoring process, we divide CFG nodes into four types: entry nodes (with indegree = 0), leaf nodes (with outdegree = 0), temp nodes (introduced for a function call (lines 1-7 in Algorithm 1)), and others called normal nodes.

We first explain how to collect text position range for three CFG node lists. We initialize three empty lists: *replacePosRangeList*, *insertBeforePosRangeList*, and

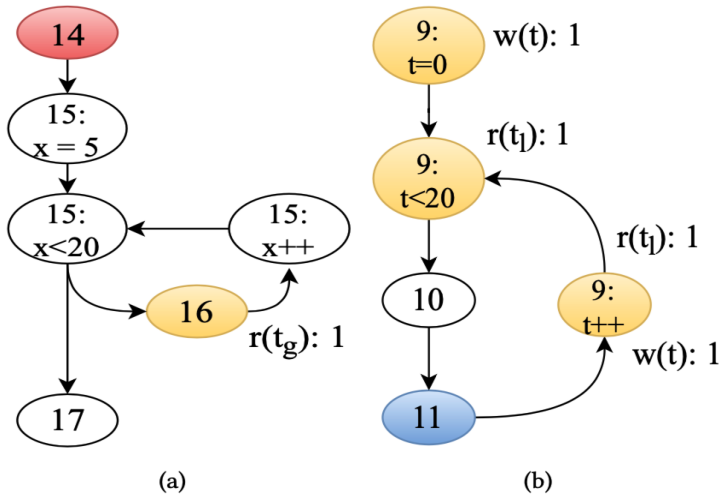

Figure 4.11 (a) The subgraph g_1 after line 28 of Algorithm 1 for f1 in Figure 4.2. Node 14 is in *list2*, and Node 16 is in *list1*. (b) The subgraph g_2 after line 36 of Algorithm 1 for f1 in Figure 4.2. Node 11 is in *list3*, and the nodes colored yellow are in *list1*.

insertAfterPosRangeList. Then we collect the text position ranges for all nodes in the CFG. We put the position range of the nodes in *list1* into *replacePosRangeList*. For entry or temp nodes in *list2*, because we cannot insert anything before them, so we put the text position ranges of their direct successor nodes into *insertBeforePosRangeList*. For other nodes in *list2*, we directly put the text position range of them into *insertBeforePosRangeList*. Similarly, for leaf or temp nodes in *list3*, because we cannot insert anything after them, we put the text position ranges of their direct predecessor nodes into *insertAfterPosRangeList*. For other nodes in *list3*, we directly put the text position range of them into *insertAfterPosRangeList*.

Next, according to each text position range in *replacePosRangeList*, *insertBeforePosRangeList*, *insertAfterPosRangeList*, we propose Algorithm 2 in Figure 4.12 to generate the optimized contract function. Lines 1-6 collect the text positions of all syntax nodes in the AST. Lines 8-12, 15-19, and 22-26 determine the corresponding AST nodes from the text positions of the CFG nodes. Especially

in lines 12-13, 18-19, and 25-26, if the text position range of a CFG node falls into the position range interval of an AST node N_a, the context syntax node C [7] containing N_a is determined in the order of priority below:

- forStatement, whileStatement, doWhileStatement
- ifStatement
- simpleStatement
- statement

For example, in Figure 4.13, the context node C for "p = y;" is the top forStatement not statement. Items that need to be added after (or before) node 11 of the CFG should be inserted after (or before) the last (or the first) token of this forStatement. For Algorithm 2 in Figure 4.12, line 20 inserts "s0 = s" before the start token of the context of an AST node, and line 27 inserts "s = s0" after the last token of the context of an AST node.

We apply Algorithm 2 in Figure 4.12 to refactor function *f1* in Listing4. The optimized contract is shown in Figure 4.4. Notably, the $list3$ for Figure 4.2 using Algorithm 1 is [11], which indicates that we need to insert "$t = t0$" after CFG node 11. But as Figure 4.13 shows, the statement for node 11 ("$p = y$") is in a loop, we should insert "$t = t0$" after the last AST syntax node "}" of the outer forStatement context, because all t in the loop will be replaced by t_0 and it is redundant to reassign t_0 to t in each iteration.

4.5 Performance Evaluation

4.5.1 Experimental Design

The smartbugs [41] dataset contains 47398 unique Solidity files. Because usually more than one contract exists in one Solidity file, one contract may not contain only one state variable. Therefore, we randomly choose 13000 Solidity contracts from the smartbugs [41] dataset to conduct our experiments. The system

[7] https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4

INPUT: The contract function, a contract state variable *s*, and three text ranges lists *replacePosRangeList*, *insertBeforePosRangeList*, *insertAfterPosRangeList*.
OUTPUT: The optimized contract function.

```
1:  get the abstract syntax tree (AST) for function f
2:  initialize the mapping MapForAstNodes from the AST node to
    the text position range in f as an empty dictionary
3:  for each node N in the AST do
4:      startPos_a = text position where N starts
5:      endPos_a = text position where N ends
6:      add (N, [startPos_a , endPos_a ]) to MapForAstNodes
7:  for each [startPos_c, endPos_c] in replaceNodePosList do
8:      for each node N in the AST do
9:          startPos_a = MapForAstNodes[N][0]
10:         endPos_a= MapForAstNodes[N][1]
11:         if startPos_a <= startPos_c and endPos_c <= endPos_a then
12:             find the context C containing N
13:             replace all occurrences of "s" in C with "s0"
14: for each [startPos_c, endPos_c] in insertBeforeNodePosList do
15:     for each node N in AST do
16:         startPos_a= MapForAstNodes[N][0]
17:         endPos_a= MapForAstNodes[N][1]
18:         if startPos_a <= startPos_c and endPos_c <= endPos_a then
19:             find the context C containing N
20:             insert "s0 = s" before the start token in C
21: for each [startPos_c, endPos_c] in insertAfterNodePosList do
22:     for each node N in the AST do
23:         startPos_a = MapForAstNodes[N][0]
24:         endPos_a = MapForAstNodes[N][1]
25:         if startPos_a <= startPos_c and endPos_c <= endPos_a then
26:             find the context C containing N
27:             insert "s = s0" after the last token in C
28: return the optimized contract function
```

Figure 4.12 Algorithm 2: Generate the optimized contract function.

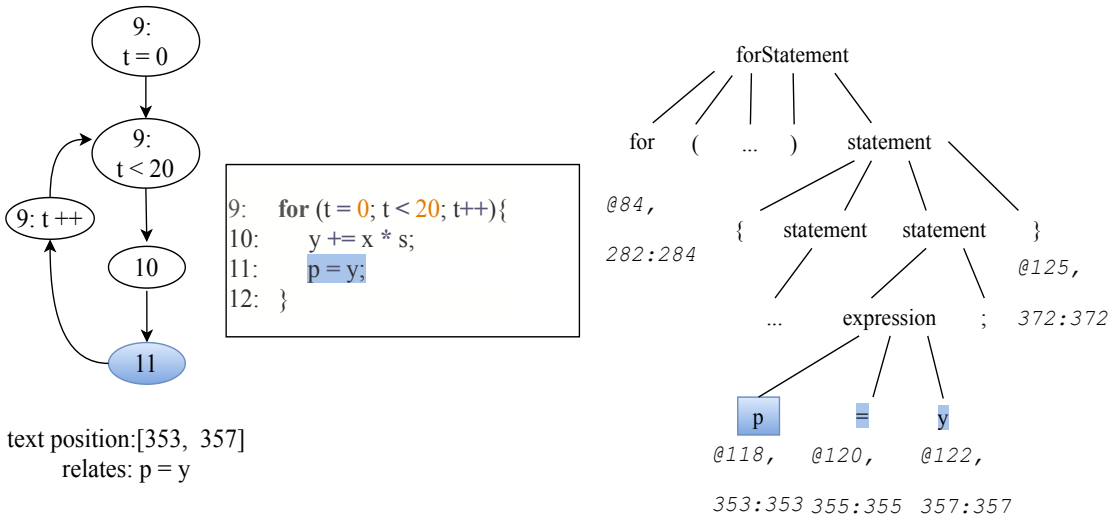

Figure 4.13 From left to right, the control flow graph, source code, and abstract syntax tree are depicted for lines 9-12 in Figure 4.2. The different representations for line 11 are colored in blue.

configuration includes MacOS Mojave 10.14.6 equipped with a 2.6 GHz Intel Core i7 CPU and 32 GB 2400 MHz DDR4 SDRAM. We use Slither [42] version 0.6.13 to build the control flow graph and the parser generated by Antlr4 [84] based on Solidity syntax [8] to produce the abstract syntax tree. We implement Algorithm 1 (in Figure 4.9) in Python and Algorithm 2 (in Figure 4.12) in Java because Antlr4 is written in Java and many auxiliary materials are available for us to use.

Storage gas estimation for a smart contract To compare the gas cost differences between under-optimized and optimized contracts, because there is no available gas model at the Solidity level, we define an abstract cost model as below.

$$Gas(StateVar, path) = T_r * Gas(SLOAD) + T_w * Gas(SSTORE) \tag{4.1}$$

[8] https://github.com/solidityj/solidity-antlr4/blob/mASTer/Solidity.g4

$$Gas(path) = \sum_{i=1}^{n} Gas(StateVar_i, path) \tag{4.2}$$

$$Gas(function) = \sum_{j=1}^{m} Gas(Path_j) \tag{4.3}$$

$$Gas(contract) = \sum_{k=1}^{q} Gas(function_k) \tag{4.4}$$

$$Gas_{optimized} = Gas_{contract} - Gas_{optmized\ contract} \tag{4.5}$$

Given the CFG of a contract function, a path is a possible execution flow from the entry node to the leaf node. If there is a loop in the path, then we only count the nodes in the loop once. Equation 4.1 computes the storage access gas cost for a state variable s in path p. We use T_r and T_w to represent the number of times s in p is read and written, respectively. Since the read and write operations of a state variable s are translated to the SLOAD and SSTORE opcodes, respectively, we calculate the gas cost for s in p as the product of T_r and the gas cost of the SLOAD opcode plus the product of T_w and the gas cost of the SSTORE opcode. *Gas(SLOAD)* and *Gas(SSTORE)* are set to 800 and 5000, respectively. Suppose there are n state variables in the contract. Equation 4.2 counts the total gas cost for storage in the function path p for n state variables. Then, Equation 4.3 takes the summation of the gas costs for storage in all paths as the gas cost for the contract function. Equation 4.4 states that the gas cost of a contract is the summation of the gas costs of all functions. Last, the optimized gas cost is defined as the gas cost of the original contract minus the gas cost of the optimized contract in Equation 4.5. Our abstract gas model is generally rational because the storage costs compose the majority of the transaction execution cost.

Each execution path in the original contract should correspond to an execution path in the optimized contract. In an optimized contract, for a state variable s, any path from the entry node to the leaf node in the CFG reads and writes s the least number of times. Therefore, in principle, the abstract gas cost of each execution path in the original contract should be not greater than the abstract gas cost of the corresponding execution path in the optimized contract.

4.5.2 Experimental Results

We apply Algorithm 1 to collect smells, i.e., CFG nodes that need to be refactored. The results are listed in the left pie chart of Figure 4.14. We find that 5734 contracts have at least one smell to be refactored, occupying 44% of the total 13000 contracts. This means that almost half of the contracts have inefficient gas use conditions. Because one contract may contain more than one state variable, we randomly select 1330 contracts from 5734 inefficient contracts to perform optimization. As shown in the right pie chart of Figure 4.14, 434 contracts are successfully optimized, and these compose 33% of all inefficient contracts. Figure 4.15 shows that 485 functions are successfully optimized. The contracts or functions that are not optimized mainly fail for two reasons: 1. Slither fails to generate a control flow graph due to inconsistent solc compiler versions[9]. 2. Some contracts' syntaxes are not covered in the Solidity syntax file used in our experiments.

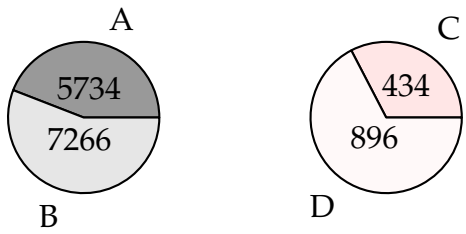

Figure 4.14 The number of contracts that: A: contain at least one refactoring nodes; B: do not need to be refactored; C: successfully optimized; D: failed optimization.

Analysis of money saved Now, we calculate the money that can be saved using our optimization technique. As shown in Table 4.3, for 434 inefficient contracts, 485 functions are successfully optimized. By comparing the abstract gas costs (defined in Section 5.5.1) for original contracts with those for optimized contracts, we find that 17,874,400 gas are saved. Considering that the gas price is 46.78 GWei and 1 GWei can be exchanged for 0.00000060 USD at the time of this writing, the invocation of our optimized contracts can save 501 USD. On

[9]We use solc 0.4.26 in our experiments, but some contracts require higher solc versions.

average, 1.03 USD and 1.16 USD can be saved for transactions with one function and one contract, respectively. Considering that more than 7 million smart contracts are currently deployed on the Ethereum mainnet as of October 2020, and we only optimize 434 contracts, the money that can be saved by optimizing 7 million contracts should be significantly larger than the amount of money saved by optimizing 434 contracts.

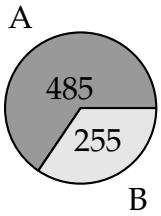

Figure 4.15 A: the number of functions that are successfully optimized; B: the number of functions that failed optimization or yielded errors.

Table 4.3 The saved storage gas cost for the successfully optimized contracts in the experimental results. (At the time of this writing, the gas price is 46.78 GWei, and 1 GWei is equivalent to 0.00000060 USD)

Total gas saved	17,874,400 ($501)
Functions involved	485
Average gas saved for one function	36,854 ($1.03)
Contracts involved	434
Average gas saved for one contract	41,184 ($1.16)

4.5.3 Discussion

We do not compare our optimization results with those of other works. There are two reasons for this. First, there is no available gas cost model at the Solidity code level. We are the *first* to build an abstract gas cost model at this level. In addition, the existing experimental results regarding gas optimization are from Chen et al.'s work [33]. They recorded the execution traces for all historical transactions in Ethereum and replaced the gas-inefficient bytecode sequences in these execution traces with gas-efficient sequences. They found that approximately 1,520 USD can be saved for 1,500 smart contracts, and on average, 1.01 USD can be saved for one contract. However, since one contract function might be called many times, the money saved using our optimization should be far more than 1.16 USD for one contract.

Moreover, the soundness of our optimization technique is based on the soundness of all the tools used, such as Slither, Antlr and solc. Any failure by these tools can influence our optimization results. Manual examination shows that less than 15% of our optimized contracts are false positives, most of which are due to the syntaxes of these contracts not being covered by Solidity syntax.

4.6 Limitations and Future Work

Our approach cannot optimize functions that have a function call, but the called function writes state variables because our current use-def relation is analyzed in the same function. We will dispose of this optimization by applying a inter-procedural use-def relation analysis in future work.

We only consider the state variables of elementary types (such as int, uint, bool and address) in our experiments. One read (or write) operation with this kind of state variable is generally translated to only one SLOAD (or SSTORE) opcode. However, for compound types, such as dynamic arrays, the amount of SLOAD (SSTORE) opcodes is related to the length of the array. We leave the optimization of compound state variables for future work. Actually, in next

chapter, we will introduce another way to optimize compound variables.

4.7 Summary

Smart contracts can encode business logic and are deployed on the Ethereum blockchain. Users can send transactions to invoke a contract function, and they are charged for transaction fees, the costs of which depend on the operations required to execute the contract functions. Therefore, smart contracts with inefficient codes waste money. It is essential to perform gas optimization on smart contracts. In this chapter, we propose an approach to automatically optimize contract functions at the Solidity level so that each execution path of the contract functions can read or write the state variables with the least times. We develop three algorithms to achieve this approach. The experimental results show that (1) 44% of contracts have inefficient gas use conditions, and (2) our approach can, on average, save 1.03 USD and 1.16 USD for transactions with one function and one contract, respectively.

Chapter 5

Gas Optimization via Array Bound Checks Reduction

Previous work found that users might be over-charged for gas-inefficient patterns in smart contracts and proposed different approaches to optimize contracts. However, their approaches mostly work on data of basic types (e.g., int, bool), and it is not clear how to minimize the gas costs for compound data structures like arrays, although we observed that arrays are mostly used in smart contracts involving loops.

In this chapter, we present a tool GotBucks to automatically optimize gas-costly array accesses in smart contracts. The following sections are organized as follows. We describe our main idea and challenges in Section 5.1. Besides, we provide a running example in Section 5.2 and apply our approach to optimize the example contract. Then, we recall some preliminary knowledge in Section 5.3, including the Solidity and assembly languages for writing smart contracts. After that, we elaborate the details of our approach in Section 5.4 and evaluate our idea in Section 5.5. Last, we compare our approach with existing work in Section 5.6 and summarize this chapter in Section 5.7.

5.1 Main Idea and Challenges

Our insight is to remove the gas costs induced by unnecessary array bound checks in EVM. In the case of the Solidity language specification, to avoid illegal out-of-bound threats, each array access requires a run-time bound check to promise that the referenced index is within the declared size. i.e., between 0 and array.length − 1. To achieve the bound check, EVM explicitly compares the array index and array.length. Nevertheless, this might introduce unnecessary gas costs. First, if the upper bound for the referenced index can be statically determined as less than array.length, the bound check in EVM is redundant and should be eliminated. Moreover, for a dynamically-sized state array, the array.length is individually stored in a slot in storage. Hence, EVM fetches the length from the storage for each bound check. Note that in addition to the function call and contract creation related opcodes, **SLOAD**(read from storage) and **SSTORE**(write to storage), charge the highest gas cost [1] among all opcodes. Therefore, if the array length is unchanged for several array access, it is cheaper to read the length from storage once for the first bound check, and store the length to stack, so that the later bound check can reuse the length instead of reading it from storage.

In this chapter, we find two novel gas-inefficient patterns concerning bound checks for arrays that have not been previously identified, and devise an approach to replace these gas-inefficient array access patterns with more efficient and functionally-equivalent ones in assembly code.

We briefly describe these gas-inefficient patterns and our countermeasures below. We also elaborate them using a running example in the next section. The first gas-inefficient pattern we found is *redundant bound check*, including the following three cases:

- Single redundant check: if the value range of referenced array index is determined to be less than array.length, then this check can be eliminated.
- Group redundant check: for multiple bound checks, if an index expression is not greater than the one for a prior check, then this check can be omitted.

[1] In EIP-2200, SLOAD charges 800 gas, and SSTORE costs 5000 or 20000 gas.

- Loop-related redundant check: array.length is loop invariant and the maximum of array index in the loop is known, then the check in the loop can be lifted out of the loop.

Whenever the array bound check is redundant, instead of eliminating the bound check in low-level opcode streams or modifying the EVM, we construct assembly code to access array elements. The second gas-inefficient pattern is *redundant length*, which refers to the duplicate reading array.length from storage for accessing dynamic arrays. We collect array accesses that share the same array.length and load length from storage at their common starting points, then rewrite the array accesses in assembly.

Nevertheless, there are two technical challenges to fix these gas-insufficient patterns. First, it is very difficult to manually construct assembly code to access array elements, because it requires a deep understanding of the EVM storage mechanism, and it is hard to promise the correctness of hand-written assembly code. To address this issue, we provide a novel approach to automatically fix the gas-insufficient smart contracts. Specifically, we design different reliable templates to optimize the smart contracts at the assembly level, which not only fosters our optimization but also enables developers to access array assembly to avoid bound check. Second, the gas cost model is defined in low-level opcode level, therefore the gas differences for the high-level origin and optimized contract programs cannot be directly computed. For this, we establish an evaluation method that compares the gas difference for two contracts on the Solidity level. First, we collect all historical transactions from the Ethereum mainnet for original contracts and then send them to the original and optimized contracts separately on the local testnet. After that, we compare the gas used for two different transactions for the origin and optimized contracts. Due to the address differences on mainnet and testnet, for all transactions, we implement address mappings from mainnet to local testnet.

5.2 A Running Example

In this section, we provide an example contract defined in Solidity, which is shown in Figure 5.2. The contract[2] *Game* contains four state variables (*players, totalBets, scores, startBlock*) and two functions (*playGame*, *countResult*). The *startBlock* is the block number which includes the deploy transaction for the contract. Each user can invoke function *playGame* to put their bet to *totalBets* and intended score to *scores* list (Lines 10-11). Then the user's address and remained balance deducted by bet are stored to a new player then push it to *players* list (Lines 12-13). Users can also send transaction to call function *countResult* (see codes enclosed by green frame in Figure 5.1). If the current block number is greater than *startBlock* by 1, the *totalBets* are divided into *rewardNum* equal portions (Line 7), then the portion is added to *rewardNum* players' *balance* and the contract transfers 1 ether[3] to players accounts (Lines 9-18) if the scores for them are greater than *divideScore*, which is the score designated by the transaction sender (Line 5).

The arrays *scores* and *players* are accessed multiple times in function *counterResult* and we observe that it is gas-inefficient. By default, each array access *scores[i]* or *players[i]* requires a run-time bound check to promise that the index is less than *scores.length* or *players.length*. However, many bound checks are not necessary and waste a lot of gas. First, concerning the loop-related redundant check pattern, for the loop in Lines 9-18, the maximum of array index *i* is *rewardNum* and the lengths of arrays *scores*, *players* are not changed within loop, it will costs less gas if replace all bound checks by a check *rewardNum* <*scores.length* before the loop[4], since generally a loop might iterate many times and a check before the loop is only executed once. Additionally, in terms of the second pattern we proposed, reading *score.length* is expensive because it is stored in EVM storage, so it is cheaper to read *score.length* once for two different bound checks (of *scores[i]* in Line 5 and a check *rewardNum* <*scores.length* before the loop in Lines 9-18).

[2]The full contract code is in Listing A.1 of Appendix.

[3]Ether (ETH) is the cryptocurrency in Ethereum network.

[4]The *scores.length* and *players.length* are equal according to function *playGame*.

```
1  function countResult(uint i, uint rewardNum) {
2    if(block.number - startBlock > 1){
3      uint256 _totalBets = totalBets;
4
5      uint256 divideScore = scores[i];
6
7      uint256 meanBet = _totalBets/rewardNum;
8
9      for(i=0;i<rewardNum;i++){
10
11       if(scores[i] > divideScore){
12         scores[i] = 1;
13
14         players[i].balance += meanBet;
15         players[i].addr.transfer(1 ether);
16         _totalBets -= meanBet;
17       }
18     }
19     totalBets = _totalBets;
20   }
21 }
```

```
4   //declare new variables, like uint scores_slot
5   assembly{
6     scores_slot := scores.slot
7     scores_len := sload(scores_slot)
8     players_slot := players.slot
9   }
10  uint firstSElePos = uint(keccak256(abi.encode(
    scores_slot)));
```

```
21  if(rewardNum > scores_len) revert();
22  uint firstPElePos = uint(keccak256(abi.encode(
      players_slot)));
23  Player storage req;
24  for(uint16 i=0;i<rewardNum;i++){
25    same as line 12-18 in the lower left code
26    if(scores_i >= divideScore) {
27      assembly{
28        let mask := shl(indexInWord,0xffff)
29        let newValue := or(and(resultWord,not(mask)),
                                  and(shl(indexInWord,1),mask))
30
31        sstore(add(firstSElePos, wordIndex),newValue)
32        req.slot := add(firstPElePos,mul(i,2))
33      }
34      req.balance += meanBet;
35      req.addr.transfer(1 ether);
36      _totalBets -= meanBet;
37    }
38  }
```

```
11  if(i >= scores_len) revert();
12  assembly{
13    base_bits := mul(i,16)
14    wordIndex := div(base_bits, 256)
15    resultWord := sload(add(firstSElePos, wordIndex))
16    indexInWord := mod(base_bits, 256)
17    scores_i := shr(indexInWord,resultWord)
18  }
19  uint256 divideScore = scores_i;
```

Figure 5.1 The original function *countResult* is within green frame. The optimized function is constructed like this: insert the upper right code before Line 5, replace Line 5 and Line 9-18 with lower left and right code, separately.

```
1  contract Game{
2    Player[] public players;
3    struct Player {address addr; uint256 balance;}
4    uint256 public totalBets;
5    uint16[] public scores;
6    uint public startBlock = block.number;
7
8    function playGame(uint16 _bet, uint16 _score,
9           uint256 _balance) {
10    totalBets += _bet;
11    scores.push(_score);
12    players.push(
13    Player({addr:msg.sender,balance:_balance-_bet}));
14   }
15   function countResult(uint i1,uint rewardNum){...}
16 }
```

Figure 5.2 The example contract.

We construct the optimized function *countResult*[5] in Figure 5.1. The code surrounded by a green frame is the original function. First, we insert the upper right code before Line 5, which load *scores.length* and store it to a local variable *scores_len*. Then we replace Line 5 with the lower left code, which does the bound check and access *scores[i]* in assembly way. We will explain the details of the assembly in Section 5.4-B. Last, Line 9-18 is substituted by the lower right code which only contains one bound check (*rewardNum* <*scores.length*) before the loop.

To verify the effectiveness of our approach, we deploy two contracts with an original and optimized function *countResult* separately on the testnet, and invoke them with different *rewardNum* values to control loop iteration times ranging from 2 to 20. The gas saved is depicted in Figure 5.3 [6], which indicates that the more iteration times of the loop in *countResult*, the more gas we can save.

[5]The full optimized contract code is in Listing A.2 of Appendix.

[6]We generate this in https://anonymous.4open.science/r/943D.

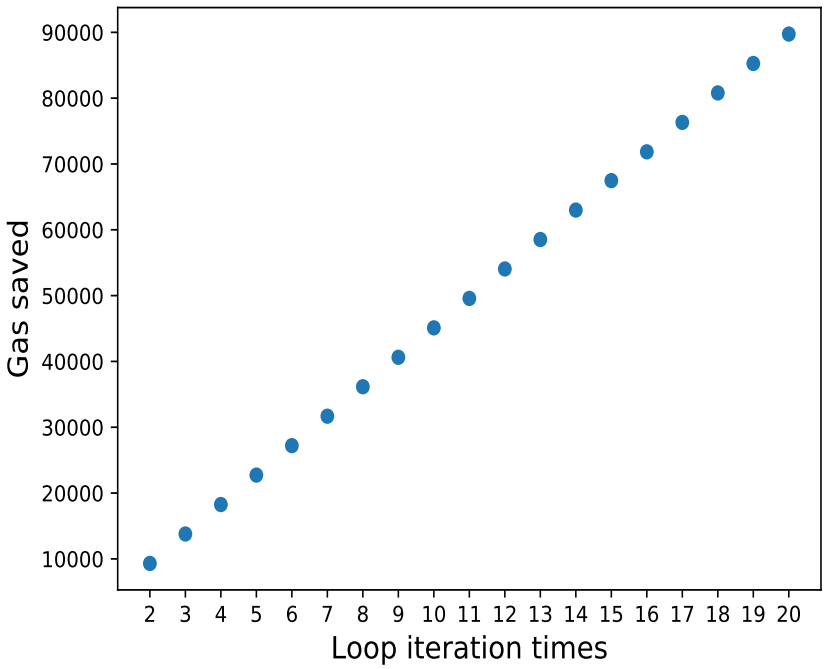

Figure 5.3 The gas saved for different loop iteration times in function *countResult*.

5.3 Preliminaries

In this section, we first recall different data locations in EVM. Then, we briefly introduce Solidity and assembly language for writing smart contracts.

5.3.1 EVM Data Locations

The EVM has four data locations to store data: storage, memory, stack, and calldata. Stack and memory are volatile and storage is persistent. Calldata cannot be modified and mainly stores function arguments. Data in memory is linear stored and addressed at the byte level. The abstract architecture of

memory is shown in Figure 5.4[7]. Storage applies a key-value store mapping 256-bit words to 256-bit words, and its structure is depicted in Figure 5.5. Stack contains 256-bit word items with a maximal depth of 1024.

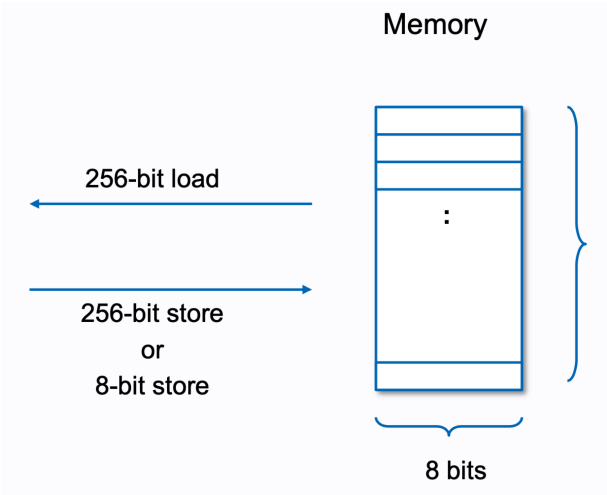

Figure 5.4 Memory structure in EVM.

5.3.2 Solidity and Assembly

Solidity

Although other languages like Vyper[13], Flint [95] exist, Solidity is still the most prevalent language for developing smart contracts on Ethereum. There are two kinds of variables allowed in smart contracts: state variables and local variables. Solidity is a statically typed scripting language that requires the type of each variable is declared. Besides value types as booleans and (u)int, Solidity allows compound types like arrays. In Solidity, an array can be of either a fixed size or dynamic size. Arrays can be stored either in storage or memory. Array elements can be of any available type. By default, to avoid illegal out-of-bound

[7] https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf.

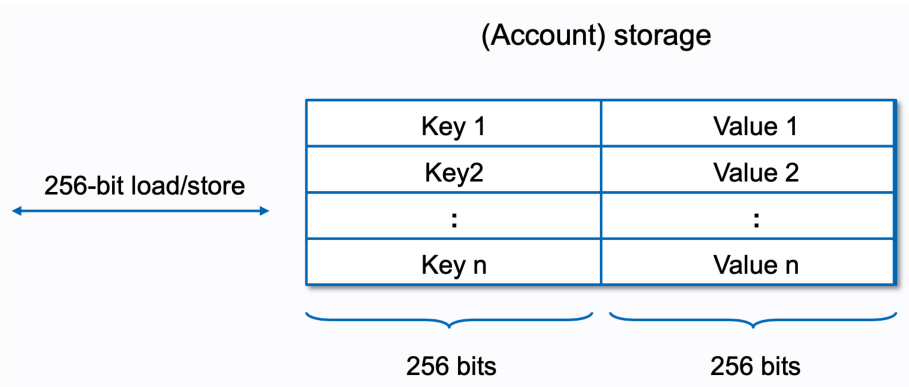

Figure 5.5 Storage structure in EVM.

threats, each array access requires a run-time bound check to promise that the referenced index is within the declared size. i.e., between 0 and array.length-1.

Assembly

To provide a more fine-grained way to access EVM, a low-level assembly language called Yul [8] is proposed to be used together with Solidity. Yul provides many built-in functions corresponding to EVM opcodes, and some of them are listed in Table 5.1. Each assembly instruction operates on words of 32 bytes. For example, the *sload(p)* instruction will push to stack the 32 bytes contents stored in storage slot *p*. The storage[p] refers to the contents at slot **p**.

5.4 Our Approach

The workflow of our approach is depicted in Figure 5.6. Given a contract in Solidity, we first extract its control flow graph (CFG) and search for the proposed gas-inefficient patterns to collect redundant bound checks (Subsection 5.4.1). Then we build assembly code to access the array to avoid bound

[8] https://docs.soliditylang.org/en/v0.5.5/yul.html

Table 5.1 Some built-in functions of Yul language.

Instruction	Explanation
add(x,y)	x+y
mul(x, y)	x * y
shl(x, y)	logical left shift y by x bits
shr(x, y)	logical right shift y by x bits
mload(p)	mem[p. . . (p+32))
mstore(p, v)	mem[p. . . (p+32)) := v
sload(p)	storage[p]
sstore(p, v)	storage[p] := v

checks (Subsection 5.4.2) and transform original contract to an optimized one (Subsection 5.4.3).

5.4.1 Collect Redundant Bound Check

We identify two kinds of gas-inefficient patterns concerning bound checks for arrays in smart contracts, i.e., redundant bound check and redundant length. Next, we define each pattern and explain their detection methods.

Redundant Bound Check

We distinguish three situations where a bound check can be removed according to the causes for the redundant check.

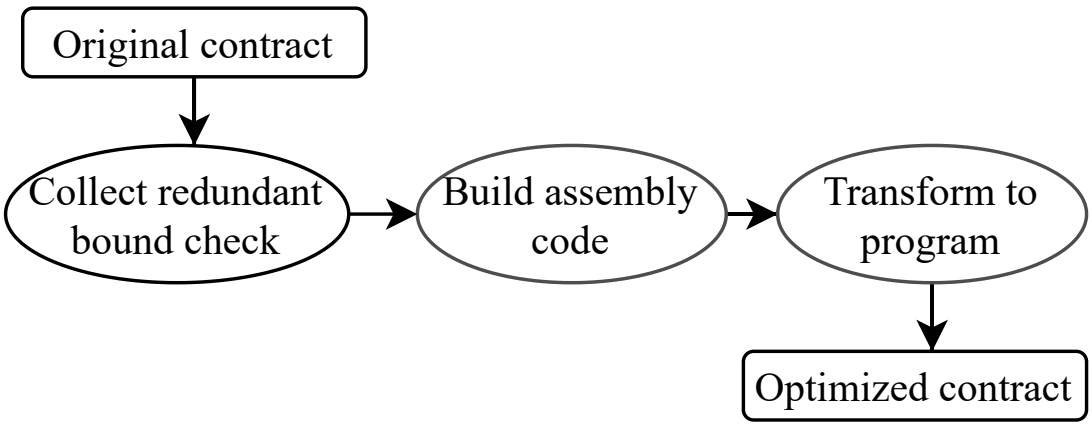

Figure 5.6 The workflow of our approach.

Group Redundant Check For an array bound check C, if exist another check C' such that either $C' = C$ or $C' \Rightarrow C$ (i.e., C' implies C), then C is redundant and can be eliminate to save gas. For example, the function *func1* in Figure 5.7 includes four bound checks (twice i¡$a.length$ and twice i+j ¡ $a.length$). Since i, j, and $a.length$ are unchanged in *func1*, the second occurrences of bound checks for $a[i]$ and $a[i+j]$ can be eliminated. Moreover, $i+j$ is always greater than i as j is an unsigned integer, so $i+j < a.length$ implies $i < a.length$. Therefore, the four bound checks can be replaced by one bound check $i+j < a.length$ before if statement (Line 3).

```
1 uint256[] a;
2 func1(uint i, uint j) {
3    if(a[i] > a[i+j])
4      a[i] = a[i+j];
5 }
```

```
6 func2(uint maxIndex) {
7   for(i=0;i<maxIndex;i++)
8     a[i] = a[i+1];
9 }
```

Figure 5.7 The func1 and func2 contain group and loop redundant checks, respectively.

To detect group redundant checks in Solidity, we apply the forward data-flow analysis algorithm developed by Gupta [50]. Their main idea is to collect

reachable bound checks for each program point. Then, for a check C, if there is a check C' can reach the point immediately before C, such that either $C' = C$ or $C' \Rightarrow C$, then C can be eliminated. Readers can refer to [50] for more details.

Loop-related Redundant Check For a bound check C is in a loop, if array.length is a loop invariant and the maximum of array index in the loop is known, then the check in the loop can be lifted out of the loop. Consider the function *func2* in Figure 5.7, the length of array *a* is unchanged within the loop, the index for bound check ranges from 0 to *maxIndex*. Solidity compiler will produce $maxIndex \times 2$ times bound check, which can be replaced by one bound check $maxIndex < a.length$ before the loop, because if $maxIndex < a.length$ succeeds, all accessed array indexes within the loop must less than $a.length$.

We reuse the loop propagation algorithm proposed in [50] to extract redundant check in loops. Their algorithm contains three procedures: 1. identify candidate bound checks in loop, such as loop invariant checks; 2. hoist candidates checks from conditionally executed blocks to unconditionally executed blocks; 3. propagate all candidate(/redundant) bound checks out of the loop.

Single Redundant Check If the value range of referenced array index is determined to be less than array.length, then this check can be eliminated. For example, a bound check *a[a.length-1]* is unnecessary for any non-empty array. We leave the detection of this kind of check as future work. The candidate approach is using range analysis to get the range of referenced index and compare it with array.length.

Redundant Length

Even there are no relationships among indexed of different bound checks, we propose a new way to optimize gas costs for array access. Our insight is that repetitive reading array.length from storage is gas-inefficient if array.length is not changed. Note that the length of a state array with the dynamic size is individually stored in a slot in storage. So EVM fetches the length from the

storage for each bound check. Note that in addition to the function call and contract creation related opcodes, **SLOAD**(read from storage) and **SSTORE**(write to storage), charge the highest gas cost [9] among all opcodes. Therefore, if the array length is unchanged for several array access, it is cheaper to read length from storage once for the first bound check, and store the length to stack, then later bound check can reuse the length instead of reading from storage. Consider the function *func3* in Figure 5.8, the referenced indexes for bound check of state array a are $i-j$, $func4(m,n+1)$, $j-i$, $func4(m,n+1)$, $i+j$, and $func4(m+1,n)$. (The $func4$ does computations on m, n and returns an integer.) The func3 doesn't contain any patterns mentioned in the last section because the size relation between these subscripts is not fixed. Nevertheless, *a.length* is same for all bound checks if $func4$ don't push to or delete elements from array a. To save gas, we can read *a.length* from storage once in Line 6, store it to a local variable (stored in the stack), and compare each subscript with the local variable.

```
1  uint256[] a;
2  func3(int i, int j,
3      uint m, uint n) {
4    if(i > j)
5      revert();
6
7    if(m > n){
8      a[i-j] = a[func4(m, n+1)];
9      a[j-i] = a[func4(m, n-1)];
10   }
11   else{
12     a[i+j] = a[func4(m+1,n)];
13   }
14 }
15 func4(uint m, uint n){...}
```

Figure 5.8 The *func3* includes redundant length checks.

To avoid extra gas costs and load array.length with least times, we develop the Algorithm 1 in Figure 5.9 to collect nodes set L where read array length

[9] In EIP-2200, SLOAD charges 800 gas, and SSTORE costs 5000 or 20000 gas.

from storage. The inputs are the CFG of a gas-inefficient function and the state array name *a*. First, we initialize nodes set *L* as an empty set. Then for each node *N* in CFG, if *N* contains a function call, push to or delete from *a*, which means the length of array *a* is changed in node *N*, we create a temporary node T in CFG and replace all outer edges of N like (N $\rightarrow$ P) with (T $\rightarrow$ P) (Line 2-8). This may split the CFG into different subgraphs and nodes in each subgraph share the same array length. Starting from line 9, we analyze each connected subgraph. The entry and leaf in line 10 refer to nodes without any in-edge and out-edge, respectively. In lines 10-12, we remove all nodes from subgraph *g* along the paths that don't access the length of array *a*. Note that the bound check conducted for reading or writing to *a* needs to access *a.length*. After this, any path starting from any entry node in *g* needs to access *a.length* at least once, so we put all entry nodes to *L*. Later, we only read *a.length* from storage before those entry nodes instead of reading *a.length* for all bound checks. Suppose the CFG is a directed acyclic graph and contains **V** nodes and **E** edges, the time complexity of Algorithm 1 is $\mathcal{O}(\mathbf{V} \times \mathbf{E})$. The most time-consuming part is in Line 9-13, where finding all paths needs $\mathcal{O}(\mathbf{V} \times \mathbf{E})$ time.

By applying Algorithm 1 to the cfg (Figure 5.10) of *func3*, *L* is [7], which means that we only need to load *a.length* from storage before node 7 and store it to a local variable. The bound checks in grey nodes (8,9,12) can reuse the local variable without reading it from storage.

5.4.2 Assembly Code Generation

Given identified two gas-inefficient patterns in the last section, we construct assembly code to access array elements. We first briefly introduce the layout of state variables in EVM storage. Then we illustrate our assembly code construction process for accessing array elements.

```
Input: The CFG of a function
Input The state array name a
Output: L, the nodes set of read a.length from storage
L = ∅
for node N ∈ the CFG do
    if N calls f() | push to | delete from a then
        if the out-degree of N != 0 then
            create a temporary node T
            for outer edge (N → P) from N  do
                remove (N → P)
                create a new edge (T → P)
for each connected sub-graph g of the CFG do
    for each path p from entry to leaf in g do
        if not (p read a & write a & use a.length) then
            remove all nodes on p from g
    add all entry nodes in g to L
return L
```

Figure 5.9 Algorithm 1: Collect nodes set in control flow graph (CFG) where read array length from storage

EVM Storage Model

Every smart contract is related to permanent storage like a very large array. The state variables are stored as key-value pairs in storage and located with the term "slot". Except for arrays of dynamic size and mappings, data is stored in a compact way starting from slot 0. For example, Figure 5.11 shows the storage layout for contract *Game* in Figure 5.2. There are four state variables: struct array *players*, uint array *scores*, and two variables with uint type. They are stored to storage in the declared order. For example, *totalBets* is stored in slot 1 because it is defined after array *players*. Note that a dynamic array first occupies a new slot of 256 bits to store its length, and the values in the array are stored consecutively from a slot that is computed using the keccak-256 hash

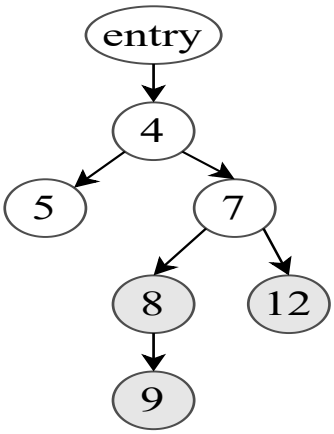

Figure 5.10 The control flow graph for *func3* in Figure 5.8. The label in each node stands for the line number. By applying Algorithm 1, the length load operations for bound check in grey nodes (8,9,12) can be replaced by a length read before node 7.

of its length. The elements in the array might share the same slot if the total size of them is no longer than 256 bits. So *players.length* and *scores.length* are stored in slot 1 and 3, respectively. The elements of *players* are stored starting at keccak-256 (SHA3 in Figure 5.11) (0). Each player occupies two slots, one for each member. For array *scores*, because each element is not longer than 128 bits, some of them share the same slot. For example, the first 15 elements of *scores* are stored sequentially in slot SHA3(2).

Read Array in Assembly

Inspired by a small trial of Leo Arias[10], we develop a general way to read an element of an array in storage or memory. For easy to use, we build a template that can be modified for different array types, which is shown in Figure 5.12. We suppose the array name is *a*. First, we need to compute the position of the first array element. For an array *a*, if it is stored in storage, the first element is

[10]https://forum.openzeppelin.com/t/playing-with-dynamic-arrays-in-assembly/1170

slot	contents			
0	players.length			
1	totalBets			
2	scores.length			
3	startBlock			
…				
SHA3(0)	players[0].addr			
SHA3(0)+1	players[0].balance			
…				
SHA3(0)+ 2*i	players[i].addr			
SHA3(0)+ 2*i+1	players[i].balance			
…	16 bits		16 bits	16 bits
SHA3(2)	scores[15]	…	scores[1]	scores[0]
…				
SHA3(2)+ k *16 / 256	…		scores[k]	…
	k*16%256			

Figure 5.11 The storage layout and contents for contract *Game* in Figure 5.2. SHA3 refers to keccak256 hash function. The contents are grouped in 256-bit words.

stored in the keccak256(p) where p is the slot number of a. If it is a memory array, it is stored closely following the memory address of a. Then in line 2, we achieve the bound check to make sure that the index i is not out of the bound *a.length*. Note that the array elements might share the same slot if the total size of them is no longer than 256 bits. Therefore, if the elements are not longer than 128 bits, to find $a[i]$, we load the word in the slot number of memory address that $a[i]$ resides (Line 3-4) and extract bits for $a[i]$ from the word using a bitwise shift operation (Line 5-6).

We explain the assembly code to read *scores[i]* within the function *countResult*, which is shown in Figure 5.1. First, statements in line 6 and line 10 of the upper right code store the position for *scores[0]* to *firstSElePos* . Then, the lower left code shows the remained processes to catch the value of *a[i]*. Concretely, line

```
1 Compute first element position
2 Check i < a.length
3 Get slot number or memory address of the i_th element
4 Load whole word from the slot or memory address
5 Figure out the bit position of i in word
6 Execute bitwise shift operation on word to get a[i]
```

Figure 5.12 A template to read the i_{th} element of array a.

11 compares *i* and *scores.length* and reverts if *i* is out of the bound. Line 13-14 compute the *wordIndex* of the slot where *a[i]* resides starting from *firstSElePos*. Especially, 16 is the bits that each array element occupies. Remember that the state variable is stored in storage, so we use *sload* to catch the whole word in *firstSElePos* + *wordIndex* and assign it to *resultWord* in line 15. Line 16 finds the index (*indexInWord*) of *a[i]* inside the *resultWord* using the *mod* instruction and line 17 shift to the right *indexInWord* bits on *resultWord* to expose the starts bits of *a[i]*. Because we declared the *scores_i* as an uint16 variables, line 17 will only extract 16 bits from the shifted *resultWord* to *scores_i*.

We now discuss some special cases for achieving the array read in assembly. First, for a memory array a, the assembly computing the position of first array element is add(a,32) because: 1) data stored in address a is *a.length*; 2) the items are closely followed a; 3) each memory address is mapped to 32 bytes data. Second, the assembly code in lower left of Figure 5.1 can be directly applied when array elements are of types like different bytes of uint and int, bool, and address. For bytes array a, we use shift left instruction shl(indexInWord, resultWord) to extract *a[i]*, because bytes array is stored in big-endian format, i.e., the most significant byte is in the smallest address, whereas numbers array is stored in little-endian format. For example, the byte array ['8', 'f', 'a', '5'] is stored as 0x8fa500...000, but the int8 array [8,15,10,5] is stored as 0x00...005af8. Third, the members of a struct array element can be accessed directly using . operator. For example, for the struct array *players* of contract *Game* in Figure 5.2, the assembly to read *players[i].balance* is shown in lower right of Figure 5.1. Specifically, line 23 defines a storage reference req, and the slot it refers is

updated in line 32 in each loop iteration. Note that each struct element occupies two slots as the total size of them is longer than 256 bits. Therefore, the start slot for *palyers[i]* is computed as firstPElePos + i*2 where the slot for *palyers[0]* firstPElePos is determined in line 22. Then each member of *players[i]* (like *players[i].balance*) can be accessed as req.memberName (like req.balance).

Write to Array in Assembly

We explain the assembly code construction for writing to array now. The general procedures are listed in Listing 5.13. Suppose we need to write *t* to *a[i]* where *a* is an array. Line 1-4 loads the data *word* from the slot number or memory address corresponding to *a[i]*. Then according to the formula in line 6, we create a new *word* as newWholeVal where only *a[i]* is updated to *t* and other bits are not changed. The value of mask is all zero excepts for all data bits to be updated (and only them) set to 1. For example, for array *scores* of contract *Game* in Figure 5.2, if we update *scores[2]* with *1*, the mask is a 256-bits number 00..00ffff00000000 in hex because *scores[2]* occupies 16 bits and the right eight zeros is related to positions of *scores[1]* and *scores[0]*, respectively. Suppose originally *scores[2]* is 4, then *scores* layout is __..__0004_____ where other bits are ignored as underscores. To compute newWholeVal, the corresponding word, ~mask(i.e., the logical negation of mask), t and mask are depicted in Figure 5.14. According to the formula in line 3, the value of newWholeVal is __..__0001_____. Note that the operations in the formula are in bit-level and will not induce integer overflow/underflow. Last, we store newWholeVal to the slot or memory position as line 7 states. The assembly in line 28-31 of the lower right code of Figure 5.1 achieves above process.

5.4.3 Source-to-source Program Transformation

To optimize the gas-inefficient patterns proposed, we design a source-to-source transformation on smart contracts. Our optimization is fully automatic on the Solidity level. First, because the pattern matching is done in the control

```
1 Compute first element position
2 Check i < a.length
3 Get slot number or memory address of the i_th element
4 Load whole word from the slot or memory address
5 Construct a mask stream mask
6 Let newWholeVal = (word & ~mask) | (t & mask)
7 Store newWholeVal to the slot or memory address of
      the i_th element
```

Figure 5.13 A template to update the i_{th} element of array a. The new value is t.

flow graph (CFG) of a contract, we collect the CFG nodes suffering from gas-inefficient patterns. We insert different labels on contract text positions corresponding to those CFG nodes to tag places where array elements are accessed by assembly code. Moreover, we tag out the places where we insert statements like computing the position of the first array element or read array length from storage. After that, we generate assembly code to read or write an array using techniques in the last section. Last, the labels in the contract are substituted using proper assembly codes.

5.5 Performance Evaluation

Based on our approach, we develop a tool named as GotBucks that can automatically optimize smart contracts for two proposed gas-inefficient patterns. Specifically, we attempt to solve the research questions (RQs) as follows.

RQ 1: Are gas-inefficient patterns prevalent in deployed contracts? In other words, how many contracts suffering from gas-inefficient patterns proposed in this chapter?

RQ 2: How useful is our developed tool GotBucks? Compared with original contracts, what percentage of contracts can be successfully optimized?

```
      word _ _ . . _ _ 0 0 0 4 _ _ _ _ _ _ _ _
     ~mask 1 1 . . 1 1 0 0 0 0 1 1 1 1 1 1 1 1
         t 0 0 . . 0 0 0 0 0 1 0 0 0 0 0 0 0 0
      mask 0 0 . . 0 0 f f f f 0 0 0 0 0 0 0 0
 ------------------------------------------------
newWholeVal _ _ . . _ _ 0 0 0 1 _ _ _ _ _ _ _ _
```

Figure 5.14 Suppose originally *scores[2] = 4* as *word* shows. We want to assign 1 to *scores[2]*.

RQ 3: How much gas can be saved using our tool?

5.5.1 Experimental Setup

Dataset for Evaluation

The Smartbugs [41] Dataset contains the source code of 47,398 unique smart contracts. We analyzed them using Slither [42] and counted the occurrences of array bound checks. The results showed that a total of 140,151 bound checks exist in 10,304 contracts and 55,820 functions. We use the contracts containing bound checks as our dataset for evaluation.

System Configuration

We conducted our all experiments on a machine running MacOS Mojave 10.14.6, with a 2.6 GHz Intel Core i7 CPU and 32 GB 2400 MHz DDR4 SDRAM. GotBucks is developed based on these tools: Slither v0.7.0, node v10.16.3, Solidity compiler solc v0.4.26 and Ganache CLI v6.12.1 (ganache-core: 2.13.1) with Constantinople

EVM [11].

5.5.2 GotBucks Implementation

We implement the two algorithms in [50] and Algorithm 1 in this chapter to find CFG nodes in contracts for two gas-inefficient patterns, respectively. (We leave the detection of single redundant checks in the future.) To conduct source-to-source transformation, we first determine the text position ranges in the contract file corresponding to CFG nodes using source mapping provided by Slither. Then according to node types, we choose proper positions to insert statements before the statement where node related to. For example, in Figure 5.10, we need to read array.length from storage before node 7. The alphabet in the position given by source mapping is "m". Therefore, we start at that position and look to the left to find the closest occurrence of "if", where we insert the length read statement. Moreover, the solc version used in our experiments is 0.4.26 because most contracts in the Smartbugs dataset can be successfully compiled with this version, but some assembly syntaxes we mentioned before are only applicable to higher solc versions so that we don't implement them in our tool. For example, the storage variables req.slot cannot be assigned to in solc v0.4.26, so that an error will be reported when an assembly statement req.slot := add(firstPElePos, mul(i,2)) occurs in contract file, whereas this syntax is allowed with higher solc versions, like solc v0.8.1.

5.5.3 Replay Transactions on Local Testnet

Because the gas cost model is defined in low-level opcode level, therefore the gas differences for the high-level origin and optimized contract programs cannot be directly computed. For this, we collected all historical transactions from the Ethereum mainnet for original contracts, and then sent them to the original

[11] Because two bitwise shifting instructions (shl and shr) are only available for Constantinople-compatible VMs, we choose Constantinople fork of Ganache-CLI.

and optimized contracts separately on the local testnet Ganache, as depicted in Figure 5.15, where the initial states of two local chains are same. Concretely, there are four steps to achieve that. First, given an original contract C_o, we crawled the hashes of all transactions to C_o from Etherscan. Second, we pulled transaction details, such as sender and input, from an Ethereum node provided by Infura [12] service. Third, due to the address differences on mainnet and testnet, for all transactions, we implemented address mappings from mainnet to local testnet. In other words, we replaced all occurred addresses of the original transaction with local addresses in Ganache. The addresses for a transaction to a contract include sender address, contract address, address in input data, and the addresses in the contract. Fourth, we sent modified transactions to original and optimized contracts, separately. By comparing the gas difference in the receipts of two kinds of transactions, we counted the gas saved in our experiments. We developed a tool to automatically achieve the above procedures.

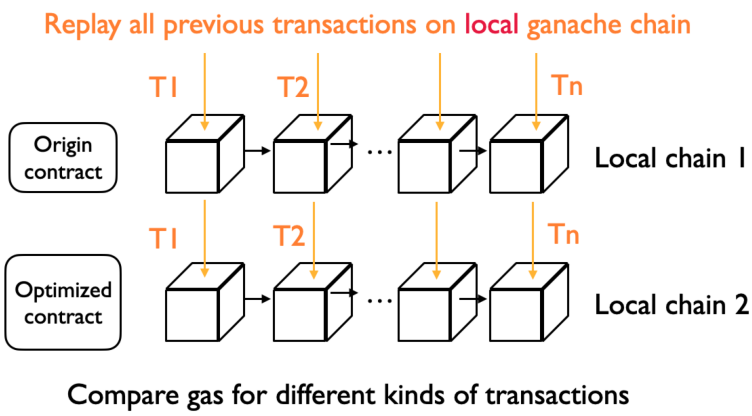

Figure 5.15 Replay transactions on origin and optimized contracts.

5.5.4 Experimental Results

We try to answer the research questions in this subsection. Especially, we detected two types of the first gas-inefficient pattern: group redundant check

[12] https://infura.io/

and loop checks. To better evaluate our results, for group redundant checks, we separately considered identical and subsume checks.

RQ1: Prevalence of Gas-inefficient Patterns

We applied GotBucks to analyze 10,304 contracts containing at least one occurrence of array access. The results are listed in Table 5.2. 8.4% of the contracts are flagged by GotBucks as containing group-identical array bound checks, 0.5% include group-subsume checks, 25% have loop redundant checks, and 34% own redundant length. From the view of bound checks, GotBucks identified that the pattern of redundant length occupies 6.9%, which more than other patterns in smart contracts.

Table 5.2 Number of bound checks, contracts, and functions containing proposed gas-inefficient patterns. (The dataset for evaluation contains totally 140,151 bound checks, 10,304 contracts and 55,820 functions.)

	Bound checks	Contracts	Functions
Group-identical	4,716 (3.4%)	862 (8.4%)	1,094 (2%)
Group-subsume	469 (0.3%)	52 (0.5%)	75 (0.1%)
Loop check	6,469 (4.6%)	2,550 (25%)	3,142 (5.6%)
Redundant length	9,617 (6.9%)	3,550 (34%)	5,460 (9.8%)

Answer to RQ1: 33.9% contracts involving array accesses contain the first gas-inefficient pattern, and 34% include the second pattern. Considering that thousands of contracts are deployed on Ethereum, proposed gas-inefficient patterns are prevalent.

RQ2: Effectiveness of GotBucks

We applied GotBucks to automatically optimize contracts with gas-inefficient patterns. Our optimization will introduce new local variables to contract, such as the variable that holds the position of the first array element. To avoid stack too deep error caused by multiple local variables in our experiments, we only optimize one pattern for state arrays at most for each contract. Generally, our optimization applies to all contracts in theory. The results are depicted in Figure 5.16. The shadowed bar represents the optimized contracts by GotBucks and the white bar shows the non-optimized contracts. For example, for the group identical pattern, 40% contracts are optimized by GotBucks and 60% are not. Note that the contracts will be conducted twice if they contain more than one pattern. We observe that for group-identical, group subsume, and redundant length, more contracts are not optimized than optimized ones. The reason is that the arrays used in non-optimized contracts are either struct arrays or memory arrays, which need a compiler of higher version and are skipped in current optimization. They will be evaluated in the future. Moreover, 57% contracts are optimized for loop check patterns, which indicates that the arrays of basic types, like int and address, are used more often than struct arrays.

Answer to RQ2: Our tool GotBucks are effective. Except for struct arrays and memory arrays (skipped for incompatible solc version), GotBucks can successfully optimize remained array types for proposed gas-inefficient patterns. Especially, for loop check pattern, GotBucks can optimize 57% inefficient contracts.

RQ3: Gas Saved Analysis

We counted the gas saved for optimized contracts by GotBucks in our experiment. We collected all historical transactions from the Ethereum mainnet for original contracts, and then replay them to the original and optimized contracts separately on testnet. The results are shown in Table 5.3.

As Hartel et al. [52] mentioned that, the environmental parameters like

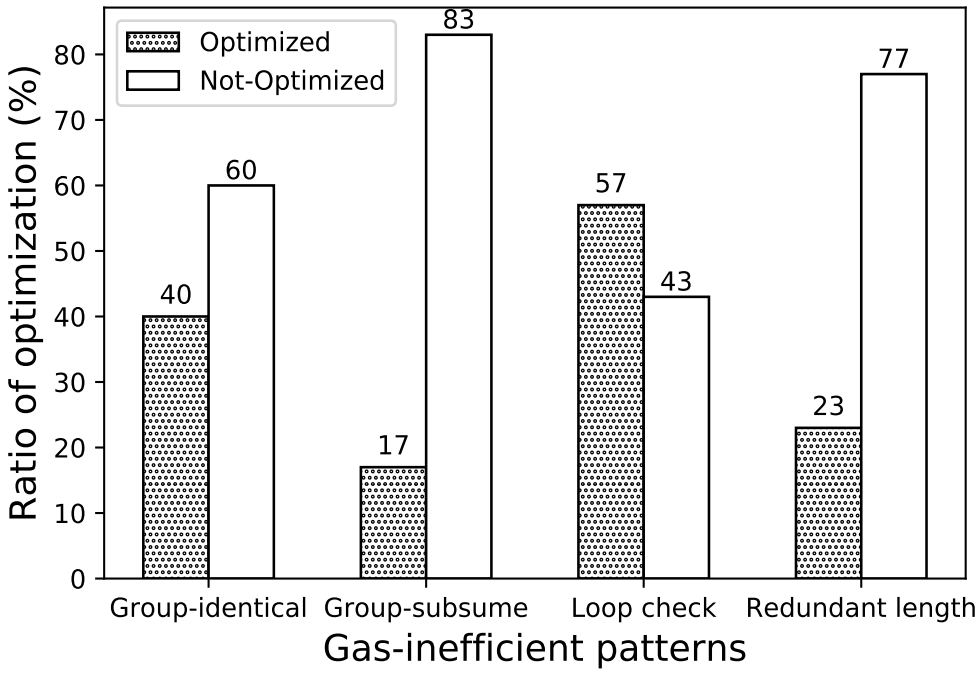

Figure 5.16 The percentage of optimized and not optimized contracts for different gas-inefficient patterns

blockhash in Ganache might not be the same as that in mainnet. Moreover, some contracts may depend on previously deployed contracts in mainnet, whereas the latter does not exist in Ganache. Therefore, the successful transactions in Ethereum mainnet might fail in our local testnet Ganache. The percentage of replay succeed contracts are 57%, 24%, and 49% for group identical checks, loop checks, and redundant length, respectively. The contracts containing group subsume checks are all replayed in failure. The gas saved is related to the successfully optimized contracts. According to the gas price at the time of writing, the saved money is 1714 dollars (4,746,178 gas) and 106 dollars (294,014 gas) by eliminating first and second gas-inefficient patterns.

We now explain why the gas saved is not so high. First, in our experiments, we only optimize one pattern for state arrays at most for each contract to avoid

Table 5.3 The saved gas cost for the successfully optimized contracts. At the time of writing, the average gas price is 155 Gwei, and 1 Gwei is equal to 0.00000233 USD.

	Group-identical	Group-subsume	Loop check	Redundant length
Optimized contracts	346	9	1,464	817
Replay failed	150	9	1,119	420
Replay succeed	196 (57%)	0	345 (24%)	397 (49%)
Gas saved	24,050	-	4,722,128	294,014
Gas saved	4,746,178 ($1714)			294,014 ($106)
Gas saved for each contract	8773 ($3.17)			741 ($0.27)

deep stack errors in contract compilation. Consider that a contract might contain more than one inefficient pattern. In actual cases, developers can choose to optimize all patterns using GotBucks in one contract and manually remove multiple local variables using different solutions. For example, they can apply struct structure to encapsulate some variables together. Second, the replayed transaction might not invoke the function we optimized. Third, the transactions calling the optimized function are not so many, so that the gas saved is not so high. But think about the tens of millions of contracts deployed in Ethereum, the gas saved can be far more than that in our experiments.

Answer to RQ3: In our evaluation, 4,746,178 gas and 294,014 gas can be saved for two different patterns. GotBucks can, on average for each contract, save 3.17 USD and 0.27 USD for optimizing the first and second patterns proposed, respectively.

5.5.5 Threats to Validity and Clarifications

The soundness of our optimization technique relies largely on the soundness of all the tools used, including `Slither`, `node`, and `solc`. Additionally, the contracts we evaluated are taken from the Smartbugs [41] dataset, which might not be enough to reflect on the effectiveness of GotBucks for all contracts deployed in the Ethereum network. However, this dataset has been successfully applied to analyze the detection results of other tools in the literature. To mitigate this, we will collect more contracts via Etherscan and verify the usefulness of our tool.

Although Ethereum is planning an upgrade to 2.0 soon that rely on Proof of Stake instead of Proof of Work, we are aware that Ethereum 1.0 will not be discarded and will certainly coexist with 2.0 as one of its shards. Therefore, our approach is still valid even when 2.0 comes because gas will still be consumed by all the existing smart contracts deployed to the Ethereum 1.0.

5.6 Related Work

Now we compare our work with other related works.

5.6.1 Gas Optimization

In contrast with existing work for gas optimization, our automatic gas optimization targets array accesses. Since the patterns are orthogonal, we cannot compare our optimization results with the other optimizations directly, but in principle results are additive.

Since there has not been a gas cost model at the source level, we are the *first* to propose an evaluation approach that sends historical transactions to the origin and optimized contracts to a test net to measure the real gas difference, enabling more accurate estimation of gas optimized and could be useful for evaluating any future source-level gas optimization approaches.

Because GotBucks conducts optimization on high-level Solidity code directly, the contracts can still be further optimized by other tools on low-level opcode streams, like syrup [22] and GasChecker [33]. In other words, all the approaches working on low-level are orthogonal with ours and could save more gas upon our optimized contracts.

5.6.2 Gas Estimation

The existing approaches on gas estimation cannot provide a gas cost model at the source level as we do in this chapter.

5.6.3 Vulnerability Detection in Smart Contracts

Gas-related Vulnerabilities

Grech et al. [46] studied three kinds of gas-focused vulnerabilities: unbounded mass operations, non-isolated external calls, and integer overflow; Chen et al. [35] observed that if the costs of operations are fixed, exploitable under-priced operations may exist. David Prechtel et al. [91] showed that 96.3 thousand (57%) contracts deployed on a live Ethereum network might be affected by gasless send issues.

Although such gas-related vulnerabilities are related, they are not the focus of our approach. With our results, it is anticipated that such attacks are less likely to succeed. We leave this for a further study.

Other Vulnerabilities

A lot of vulnerability detection tools have been proposed. We only list part of them. To help the user write secure contracts, Anastasia et al. [75] designed *VeriSolid* framework to generate solidity code from the verified transition-system

based models. Oyente [70] used symbolic execution to detect several security bugs. Since these vulnerabilities may affect the correctness of the smart contracts, they will still be necessary to adopt for the smart contracts generated by our approach. We leave that as a direction of future study.

5.6.4 Bound Check on Other Languages

Java also employs runtime bound checks and a lot of work has been done to eliminate them. For example, Gupta [50] proposed identical and subsumed bounds checks and removed them. We considered these checks as the first gas-inefficient pattern. In other words, we found that these patterns also exist in smart contracts. Besides, we proposed a new pattern specific to smart contracts, i.e., redundant reading, that can be optimized by our approach to save gas. Due to the gas cost model in EVM, we find it more important to reduce the uncessary bound checks compared to traditional virtual machines.

5.7 Summary

Smart contracts with gas-inefficient codes waste money. In this chapter, we built GotBucks, a gas-optimization tool to automatically reduce the gas used for accessing arrays in smart contracts. Especially, we find two novel and distinct gas-inefficient patterns concerning array bound checks that have not been identified. Evaluated on over 10,000 smart contracts, it has been shown that: (1) our proposed gas-inefficient patterns are prevalent; (2) the gas saved are promising.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

For a transaction invoking a function in smart contracts on Ethereum, the fee that the sender needs to pay is the multiplication of gas costs and gas price. The gas price is specified by transaction senders and the same for all gas, whereas the gas costs are mainly decided by computational efforts on Ethereum Virtual Machine (EVM). This thesis investigates two widely studied types of research regarding gas on Ethereum: gas estimation and gas optimization.

Gas estimation aims to predict executional gas consumption before the transaction starts, which is very helpful to avoid out-of-gas exceptions. Different approaches have been proposed for it, but we argue that it is still nontrivial in estimating gas costs for transactions to functions involving loops. Neither static only nor dynamic only methods can estimate this kind of transaction. We solve the problem by proposing a gas estimation approach to dynamically estimate the gas based on the transaction trace. The transaction trace indicates the executional opcode sequence in EVM. The insight is that the relationship between the historical transaction traces and their gas costs can be learned to estimate the gas for new transactions. Especially, three different abstractions

of the original transaction trace are considered and fed to different machine learning models. The results show that the proposed approach is effective in gas estimation for transactions to loop functions and that a random forest can achieve the most accurate estimation.

Gas optimization refers to optimization on smart contracts so that gas costly patterns can be replaced by gas efficient ones. More gas means a higher transaction fee, therefore, gas optimization allows transaction senders to pay less fee. We find two issues for gas optimization that previous work did not notice: one for storage usage and the other for array access. For storage usage optimization, we recognize that existing work only considered loop structures but not for all possible program behaviors. To mitigate this gap, we propose an approach to automatically optimize contract functions at the Solidity level so that each execution path of the contract functions can read or write the state variables with the least times. We develop three algorithms to achieve this approach. The experimental results show that (1) 44% of contracts have inefficient gas use conditions, and (2) our approach can, on average, save 1.03 USD and 1.16 USD for transactions with one function and one contract, respectively. For the array access optimization part, we observe that none of the existing work considered the optimization for compound types, such as arrays. We present a tool GotBucks to automatically optimize gas- costly array accesses in smart contracts. The insight is to remove the gas costs induced by unnecessary array bound checks in EVM. Especially, we find two novel and distinct gas-inefficient patterns concerning array bound checks that have not been identified. Evaluated on over 10,000 smart contracts, it has been shown that: (1) our proposed gas-inefficient patterns are prevalent; (2) the gas saved are promising.

The idea behind gas optimization can sometimes help gas estimation. For example, the storage usage optimization is based on the principle that more storage operations indicate a higher gas cost. Therefore, if we already know the gas costs for a transaction **T1**, for a new transaction **T2**, we can get the approximate gas costs for **T2** by comparing the storage access times between the trace of **T1** and **T2**. Additionally, we build two methods to measure the gas differences on Solidity code level. One is to compute the gas on abstract

storage cost model (in Section 4.5.1), the other one is to replay all historical transactions on original and optimized contracts separately, and then to count the gas difference (in Section 5.5.3). So that, the saved gas costs in Section 4 or Section 5 can be reevaluated using another aforementioned gas evaluation method.

6.2 Future Work

6.2.1 Gas Estimation

In this thesis, we first 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. In the future, we will tentatively extend our work in some aspects. First, to improve the prediction accuracy rate, we plan to collect more transaction traces and extract new features from the original trace to train our gas estimation model. Moreover, although our implementation is limited to the gas estimation for transactions to loop functions, we believe that our idea is also applicable to gas estimation for functions other than loops. To verify this, we will replay transactions to any kinds of functions and collect their trace. Then we apply different machine learning algorithms to build a gas estimation model. Additionally, we consider building gas estimation models for transactions to special contracts, e.g, ERC-20 token contracts, which refers to contracts implementing the ERC-20 token standard. These contracts all provide functionalities like transferring tokens and checking the balance of an account. Concretely, we will only collect traces to transactions to these contracts and train a gas estimation model. We will conduct experiments to check whether the prediction rate can be increased for transactions to ERC-20 contracts using the newly generated model.

6.2.2 Gas Optimization

We solved two problems regarding gas optimization in this thesis: one is storage usage optimization and another is array access optimization. We would like to explore other possible optimization issues in the future. First, currently in our experiments for the array access optimization part, we evaluate arrays in storage only, next we will also verify the effectiveness of our approach on arrays in memory which could save more gas. Moreover, the array bound checks in this thesis only check the upper bounds because the Solidity compiler only generates explicit opcode streams for upper bound checks. We will try the optimization for a lower bound check once they are explicit in the next generation of the Solidity compiler. Besides, we aim to develop new techniques to save gas for EVM memory operations. As far as we know, there is no previous work targeting memory usage optimization. We realize that some work [51] can formalize the Solidity memory model, whereas they can not capture the memory change in execution. We can observe this change in replayed transaction trace, then we can encode the memory change into their proposed memory model, which might be helpful to find possible ideas for memory usage optimization.

Chapter 7

Publications

1. Chunmiao Li, Shijie Nie, Yang Cao, Yijun Yu, Zhenjiang Hu. Trace-based Dynamic Gas Estimation of Loops in Smart Contracts. IEEE Open Journal of the Computer Society, 2020.

2. Chunmiao Li, Shijie Nie, Yang Cao, Yijun Yu, Zhenjiang Hu. Dynamic Gas Estimation of Loops using Machine Learning. In International Conference on Blockchain and Trustworthy Systems (BlockSys'2020), Aug. 06-07, 2020.

3. Chunmiao Li, Yang Cao, Zhenjiang Hu, Masatoshi Yoshikawa. Blockchain-based Bidirectional Updates on Fine-grained Medical Data. In First International Workshop on Blockchain and Data Management (BlockDM 2019), Macau SAR, China, Apr 8, 2019.

Bibliography

[1] 0xbounty. https://0x.org/docs/guides. ↰ page 18

[2] Bountylist. https://solidified.io/. ↰ page 18

[3] Brownie. https://eth-brownie.readthedocs.io/en/stable/core-contracts.html. ↰ page 18

[4] Chainsecurity. https://chainsecurity.com/. ↰ page 18

[5] Erc20. https://eips.ethereum.org/EIPS/eip-20. ↰ page 18

[6] Erc721. https://eips.ethereum.org/EIPS/eip-721. ↰ page 18

[7] eth-gas-reporter. https://www.npmjs.com/package/eth-gas-reporter. Accessed: 2020-04-23. ↰ page 12

[8] gas-exactimation. https://www.trufflesuite.com/blog/ethereum-gas-exactimation. Accessed: 2020-04-23. ↰ page 12

[9] Smartdec. https://smartdec.net/. ↰ page 18

[10] Solidified. consensys.github.io/smart-contract-best-practices/bug_bounty_list/. ↰ page 18

[11] Surya. https://github.com/ConsenSys/surya. ↰ page 18

[12] Truffle. https://www.trufflesuite.com/. ↰ page 18

[13] Vyper. https://vyper.readthedocs.io. ↰ pages 18 and 78

[14] Openzeppelin. https://github.com/NuoNetwork/openzeppelin-solidity, 2018. ↰ pages 16 and 18

[15] Cardano documentation. https://docs.cardano.org/en/latest/, 2020. ↰ page 1

[16] Design patterns. https://fravoll.github.io/solidity-patterns/, 2020. ↰ page 18

[17] Hyperledger fabric. https://www.hyperledger.org/use/fabric, 2020. ↰ page 1

[18] R3 corda. https://www.corda.net/, 2020. ↰ page 1

[19] Sload, 2020. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2200.md. ↰ page 3

[20] Solc. https://docs.soliditylang.org/en/v0.7.5/using-the-compiler.html, 2020. ↰ pages 13 and 52

[21] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. 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. ↰ pages 6, 7, and 12

[22] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125. Springer, 2020. ↰ pages 8, 13, 46, 47, and 99

[23] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett. Synthesis of super-optimized smart contracts using max-smt. In *International Conference on Computer Aided Verification*, pages 177–200. Springer, 2020. ↰ pages 8, 13, and 45

[24] L. Alt and C. Reitwiessner. Smt-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 376–388. Springer, 2018. ↰ page 18

[25] S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, 2018. ↰ page 18

[26] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017. ↰ page 17

[27] S. Azzopardi, J. Ellul, and G. J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In *International Conference on Runtime Verification*, pages 113–137. Springer, 2018. ↰ pages 17 and 18

[28] e. a. Bernhard E. Boser. A training algorithm for optimal margin classifiers. 2010. ↰ page 24

[29] R. S. Boyer and J. S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM (JACM)*, 31(3):441–458, 1984. ↰ page 6

[30] T. Brandstätter. *Optimization of Solidity Smart Contracts*. PhD thesis, Wien, 2020. ↰ page 14

[31] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1335–1352, 2018. ↰ page 18

[32] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. ↰ page 23

[33] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 2020. ↰ pages 8, 45, 46, 47, 69, and 99

[34] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017. ↰ pages 8 and 13

[35] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *International Conference on Information Security Practice and Experience*, pages 3–24. Springer, 2017. ↰ pages 15 and 99

[36] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang. 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)*, pages 81–84. IEEE, 2018. ↰ pages 8 and 13

[37] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1503–1520, 2019. ↰ page 26

[38] X. Chen, D. Park, and G. Roşu. A language-independent approach to smart contract verification. In *International Symposium on Leveraging Applications of Formal Methods*, pages 405–413. Springer, 2018. ↰ page 18

[39] Consensys. smart-contract-best-practices. ↰ page 18

[40] A. Dika and M. Nowostawski. Security vulnerabilities in ethereum smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 955–962. IEEE, 2018. ↰ page 17

[41] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. *arXiv preprint arXiv:1910.10601*, 2019. ↰ pages 33, 63, 91, and 98

[42] J. Feist, G. Grieco, and A. Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019. ↰ pages 18, 25, 29, 33, 65, and 91

[43] C. K. Frantz and M. Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 210–215. IEEE, 2016. ↰ page 18

[44] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114, 2019. ↰ page 6

[45] L. Goodmani. Michelson: the language of smart contracts in tezos. *URL https://tezos. com/pages/tech. html*. ↰ page 18

[46] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018. ↰ pages 15, 37, and 99

[47] I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018. ↰ page 18

[48] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017. ↰ page 18

[49] M. Gupta. solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6. ↰ pages 13 and 45

[50] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):135–150, 1993. ↰ pages 81, 82, 92, and 100

[51] Á. Hajdu and D. Jovanovic. Smt-friendly formalization of the solidity memory model. In *ESOP*, pages 224–250, 2020. ↰ page 104

[52] P. Hartel and M. van Staalduinen. Truffle tests for free–replaying ethereum smart contracts for transparency. *arXiv preprint arXiv:1907.09208*, 2019. ↰ page 95

[53] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018. ↰ page 16

[54] Y. Hirai. Formal verification of deed contract in ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf*, 2016. ↰ page 18

[55] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017. ↰ page 18

[56] T. Hukkinen et al. Reducing blockchain transaction costs in a distributed energy market application. 2018. ↰ page 14

[57] S. Hwang and S. Ryu. Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 542–553, 2020. ↰ page 18

[58] I. T. Jolliffe and J. Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016. ↰ page 40

[59] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018. ↰ page 18

[60] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb. Android malware detection using deep learning on api method sequences. *arXiv preprint arXiv:1712.08996*, 2017. ↰ page 27

[61] T. Kasampalis, D. Guth, B. Moore, T. Serbanuta, V. Serbanuta, D. Filaretti, G. Rosu, and R. Johnson. Iele: An intermediate-level blockchain language designed and implemented using formal semantics. Technical report, 2018. ↰ page 18

[62] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: theory and practice*. CRC Press, 2017. ↰ page 56

[63] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1317–1333, 2018. ↰ page 16

[64] A. Li, J. A. Choi, and F. Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020. ↰ pages 17 and 18

[65] C. Li, S. Nie, Y. Cao, Y. Yu, and Z. Hu. Dynamic gas estimation of loops using machine learning. In *International Conference on Blockchain and Trustworthy Systems*. Springer, 2020. ↰ pages 28 and 39

[66] C. Li, S. Nie, Y. Cao, Y. Yu, and Z. Hu. Trace-based dynamic gas estimation of loops in smart contracts. *IEEE Open Journal of the Computer Society*, 1:295–306, 2020. ↰ page 28

[67] K. Li, Y. Tang, Q. Zhang, C. Xu, and J. Xu. Grub: Gas-efficient blockchain storage via workload-adaptive data replication. *arXiv preprint arXiv:1911.04078*, 2019. ↰ page 14

[68] C. Liu, J. Gao, Y. Li, and Z. Chen. Understanding out of gas exceptions on ethereum. In *International Conference on Blockchain and Trustworthy Systems*, pages 505–519. Springer, 2019. ↰ pages 5 and 16

[69] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018. ↰ page 18

[70] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016. ↰ pages 16, 18, and 100

[71] F. Ma, Y. Fu, M. Ren, W. Sun, Z. Liu, Y. Jiang, J. Sun, and J. Sun. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *arXiv preprint arXiv:1910.02945*, 2019. ↰ pages 7 and 12

[72] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi. Evm*: from offline detection to online reinforcement for ethereum virtual machine. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 554–558. IEEE, 2019. ↰ page 18

[73] D. Mao, F. Wang, Y. Wang, and Z. Hao. Visual and user-defined smart contract designing system based on automatic coding. *Ieee Access*, 7:73131–73143, 2019. ↰ page 18

[74] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, and N. Sharygina. Computing exact worst-case gas consumption for smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 450–465. Springer, 2018. ↰ pages 7 and 12

[75] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey. Verisolid: Correct-by-design smart contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 446–465. Springer, 2019. ↰ pages 16, 18, and 99

[76] A. Mense and M. Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*, pages 375–380, 2018. ↰ page 17

[77] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019. ↰ page 18

[78] J. Nagele and M. A. Schett. Blockchain superoptimizer. *arXiv preprint arXiv:2005.05912*, 2020. ↰ pages 8 and 13

[79] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. ↰ page 1

[80] 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 *International Conference on Fundamental Approaches to Software Engineering*, pages 440–455. Springer, 2009. ↰ pages xi, 29, and 31

[81] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020. ↰ page 18

[82] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663, 2018. ↰ page 18

[83] A. A. K. N'Da, S. Matalonga, and K. Dahal. Characterizing the cost of introducing secure programming patterns and practices in ethereum. In *World Conference on Information Systems and Technologies*, pages 25–34. Springer, 2020. ↰ page 18

[84] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. ↰ page 65

[85] M. H. Peng, F. Yu, and J.-H. R. Jiang. Symbolic gas vulnerability detection and attack synthesis. In *PACIS*, page 107, 2020. ↰ page 15

[86] D. Perez and B. Livshits. Broken metre: Attacking resource metering in evm. *arXiv preprint arXiv:1909.07220*, 2019. ↰ page 18

[87] D. Perez and B. Livshits. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021. ↰ page 17

[88] L. Peterson. K-nearest neighbor. *Scholarpedia*, 2009. ↰ page 23

[89] J. Pettersson and R. Edström. Safer smart contracts through type-driven development. Master's thesis, 2016. ↰ page 18

[90] G. A. Pierro and H. Rocha. The influence factors on ethereum transaction fees. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 24–31. IEEE, 2019. ↰ page 14

[91] D. Prechtel, T. Groß, and T. Müller. Evaluating spread of 'gasless send'in ethereum smart contracts. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6. IEEE, 2019. ↰ pages 15, 18, and 99

[92] C. Reitwiessner. "babbage: a mechanical smart contract language, 2019. ↰ page 18

[93] M. Rodler, W. Li, G. O. Karame, and L. Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018. ↰ page 18

[94] I. Santos, F. Brezo, B. Sanz, C. Laorden, and P. G. Bringas. Using opcode sequences in single-class learning to detect unknown malware. *IET information security*, 5(4):220–227, 2011. ↰ page 27

[95] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 218–219, 2018. ↰ pages 18 and 78

[96] C. Signer. Gas cost analysis for ethereum smart contracts. Master's thesis, ETH Zurich, Department of Computer Science, 2018. ↰ pages 7 and 12

[97] L. Stegeman. Solitor: runtime verification of smart contracts on the ethereum network. Master's thesis, University of Twente, 2018. ↰ page 18

[98] D. Suvorov and V. Ulyantsev. Smart contract design meets state machine synthesis: Case studies. *arXiv preprint arXiv:1906.02906*, 2019. ↰ page 18

[99] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632*, 2018. ↰ page 18

[100] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018. ↰ page 18

[101] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018. ↰ page 18

[102] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer. Formal specification and verification of smart contracts for azure blockchain. *arXiv preprint arXiv:1812.08829*, 2018. ↰ page 18

[103] S. M. Werner, P. J. Pritz, and D. Perez. Step on the gas? a better approach for recommending the ethereum gas price. *arXiv preprint arXiv:2003.03479*, 2020. ↰ page 15

[104] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. ↰ pages 1, 2, 5, 43, and 46

[105] V. Wüstholz and M. Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020. ↰ page 18

[106] R. Yang, T. Murray, P. Rimba, and U. Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319. IEEE, 2019. ↰ page 14

[107] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi. Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 842–853. IEEE, 2019. ↰ page 14

[108] K. Zipfel. how-to-write-smart-contracts-that-optimize-gas-spent-on-ethereum-30b5e9c5db85, 2020. ↰ page 13

Appendix A

Origin and Optimized Contract for Array Optimization

Listing A.1 The origin example contract in Chapter 5

```
contract Game {
 Player[] public players;
 struct Player {address payable addr; uint256 balance;}
 uint256 public totalBets;
 uint16[] public scores;
 uint256 public startBlock = block.number;

 constructor() public payable {}

 function playGame(uint16 _bet, uint16 _score, uint256 _balance) {
    totalBets += _bet;
    scores.push(_score);
    players.push(
```

```
14     Player({addr: payable(msg.sender), balance: _balance - _bet}));
15   }
16   function countResult(uint256 i1, uint256 rewardNum) public {
17     if (block.number - startBlock > 1) {
18       uint256 _totalBets = totalBets;
19       uint256 divideScore = scores[i1];
20       uint256 meanBet = _totalBets / rewardNum;
21
22       for (uint16 i = 0; i < rewardNum; i++) {
23         if (scores[i] > divideScore) {
24           scores[i] = 1;
25           players[i].balance += meanBet; //update balance
26           players[i].addr.transfer(1 ether);
27           _totalBets -= meanBet;
28         }
29       }
30       totalBets = _totalBets;
31     }
32   }
33 }
```

Listing A.2 The optimized example contract by our approach in Chapter 5

```
1 contract Game{
2   Player[] public players;
3   struct Player {address payable addr;uint256 balance;}
4   uint256 public totalBets;
5   uint16[] public scores;
6   uint public startBlock = block.number;
7
8   constructor() payable public {}
```

```
9  function playGame(uint16 _bet, uint16 _score, uint256 _balance) {
10   totalBets += _bet;
11   scores.push(_score);
12   players.push(
13     Player({addr:payable(msg.sender),balance: _balance - _bet}));
14 }
15
16 function countResult(uint i1, uint rewardNum) public{
17   if(block.number - startBlock > 1){
18     uint256 _totalBets = totalBets;
19
20     uint scores_slot;
21     uint scores_len;
22     uint players_slot;
23     uint16 scores_i;
24     uint256 base_bits;
25     uint256 wordIndex;
26     uint256 resultWord;
27     uint indexInWord;
28
29     assembly{
30       scores_slot := scores.slot
31       scores_len := sload(scores_slot)
32       players_slot := players.slot
33     }
34     uint firstScoreElePos = uint(keccak256(
35         abi.encode(scores_slot)));
36
37     if(i1 >= scores_len) revert();
38     assembly{
39       base_bits := mul(i1,16)
```

```
40        wordIndex := div(base_bits, 256)
41        resultWord := sload(add(firstScoreElePos, wordIndex))
42        indexInWord := mod(base_bits, 256)
43        scores_i := shr(indexInWord,resultWord)
44      }
45      uint256 divideScore = scores_i;
46      uint256 meanBet = _totalBets/rewardNum;
47
48      if(rewardNum > scores_len) revert();
49
50      uint firstPlayerElePos = uint(keccak256(
51          abi.encode(players_slot)));
52      Player storage req;
53
54      for(uint16 i=0;i<rewardNum;i++){
55        assembly{
56          base_bits := mul(i,16)
57          wordIndex := div(base_bits, 256)
58          resultWord := sload(add(firstScoreElePos, wordIndex))
59          indexInWord := mod(base_bits, 256)
60          scores_i := shr(indexInWord,resultWord)
61        }
62
63        if(scores_i >= divideScore) {
64          assembly{
65            let mask := shl(indexInWord,0xffff)
66            let newValue := or(and(resultWord,not(mask)),
67                and(shl(indexInWord,1),mask))
68            sstore(add(firstScoreElePos, wordIndex),newValue)
69            req.slot := add(firstPlayerElePos,mul(i,2))
70          }
```

```
71          req.balance += meanBet;
72          req.addr.transfer(1 ether);
73          _totalBets -= meanBet;
74        }
75      }
76      totalBets = _totalBets;
77    }
78  }
79 }
```