Kong QP, Wang ZY, Huang Y *et al.* Characterizing and detecting gas-inefficient patterns in smart contracts. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(1): 67–82 Jan. 2022. DOI 10.1007/s11390-021-1674-4

Characterizing and Detecting Gas-Inefficient Patterns in Smart Contracts

Que-Ping Kong[1] (孔雀屏), *Member, CCF*, Zi-Yan Wang[1] (王子彦), *Member, CCF*
Yuan Huang[2] (黄　袁), *Member, CCF*, Xiang-Ping Chen[3,*] (陈湘萍), *Member, CCF, IEEE*
Xiao-Cong Zhou[1] (周晓聪), *Member, CCF*, Zi-Bin Zheng[1] (郑子彬), *Senior Member, CCF, ACM, IEEE*, and
Gang Huang[4] (黄　罡), *Distinguished Member, CCF, Member, ACM, IEEE*

[1] *School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China*

[2] *School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China*

[3] *Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-sen University, Guangzhou 510006, China*

[4] *Peking University Shenzhen Graduate School, Shenzhen 518000, China*

E-mail: {kongqp, wangzy75}@mail2.sysu.edu.cn; {huangyuan5, chenxp8, isszxc, zhzibin}@mail.sysu.edu.cn
hg@pku.edu.cn

Received June 2, 2021; accepted December 3, 2021.

Abstract Ethereum blockchain is a new internetware with tens of millions of smart contracts running on it. Different from general programs, smart contracts are decentralized, tamper-resistant and permanently running. Moreover, to avoid resource abuse, Ethereum charges users for deploying and invoking smart contracts according to the size of contract and the operations executed by contracts. It is necessary to optimize smart contracts to save money. However, since developers are not familiar with the operating environment of smart contracts (i.e., Ethereum virtual machine) or do not pay attention to resource consumption during development, there are many optimization opportunities for smart contracts. To fill this gap, this paper defines six gas-inefficient patterns from more than 25 000 posts and proposes an optimization approach at the source code level to let users know clearly where the contract is optimized. To evaluate the prevalence and economic benefits of gas-inefficient patterns, this paper conducts an empirical study on more than 160 000 real smart contracts. The promising experimental results demonstrate that 52.75% of contracts contain at least one gas-inefficient pattern proposed in this paper. If these patterns are removed from the contract, at least $0.30 can be saved per contract.

Keywords smart contract, anti-pattern, detection, optimization, empirical study

1 Introduction

Blockchain is a new internetware and a smart contract is a program running on it. Due to the decentralization and tamper-proof of smart contracts, smart contracts are flexibly embedded into a variety of digital assets to help achieve secure and efficient information exchange and value transfer. It provides a tremendous opportunity for different industries and fields such as supply chain, smart homes, e-commerce, and asset management [1]. As the first blockchain system to support smart contracts, Ethereum① attracts many developers. Within five years of Ethereum's launch, more than 16 million smart contracts have been deployed on Ethereum [2]. Moreover, the market capitalization of Ethereum has exceeded $40 billion, second only to Bitcoin②.

Smart contracts are usually written in a high-level

Regular Paper

Special Section on Software Systems 2021—Theme: Internetware and Beyond

This work was supported by the Key-Area Research and Development Program of Guangdong Province of China under Grant No. 2020B010164002, and the National Natural Science Foundation of China under Grant No. 62032025.

*Corresponding Author

①https://ethereum.org/en/whitepaper/, May 2020.

②https://www.coingecko.com, Dec. 2021.

language (e.g., Solidity[3]) and then compiled into bytecode which can be executed in the Ethereum Virtual Machine (EVM). As an assembly language, the bytecode is made up of multiple opcodes. Each opcode performs a certain action on the blockchain. In order to avoid excessive consumption of resources and to ensure termination, Ethereum adopts the gas mechanism that every opcode is performed costing a certain amount of gas. The term gas is a special unit to measure the consumption of storage and computing resources. The Ethereum Yellow Paper[4] stipulates the gas consumed by each opcode. It is noting that the transaction fee is equal to the gas used multiplied by the gas price and the gas price is constant generally[5]. In other words, the more the gas used by the smart contract, the higher the transaction fee paid by the contract user to the miner. In the past three years, the average gas consumption per day is 72.4 billion units, which is about 9.77 million US dollars[6]. In addition, due to that contract developers insufficiently understand the gas mechanism or do not realize that individual coding patterns will bring unnecessary gas consumption, there is a lot of room for optimization in smart contracts. Therefore, it makes sense to perform equivalent transformations on the smart contract so that more gas-saving smart contracts can be generated on the premise of not changing the running effect of the contract. For Ethereum, optimizing the gas consumption of smart contracts can improve the efficiency of contract execution, which is conducive to the sustainable development; for the users of smart contracts, gas optimization can reduce the cost of deploying and calling contracts, which can increase the activity of decentralized application (DApp).

Existing gas optimization approaches [3–6] which are based on satisfiability modulo theories detect and remove gas-inefficient patterns at the bytecode level. They attempt to find the opcode sequences which require fewer resources and are semantically equivalent to the original ones by trying all possible opcode sequences. Optimization approaches [3–5] do not consider inefficient programming patterns brought by developers. Although the optimization approaches [6] take into account some of the inefficient programming patterns brought by developers, it can only detect and not optimize. There are some gas-inefficient smart contract snippets which cannot be optimized by existing gas optimization approaches. Fig.1(a) shows a smart contract which cannot be optimized by existing approaches and Fig.1(b) gives its efficient version. In the gas-inefficiency version, the contract declares a state variable *num* and explicitly assigns 0 to the variable, which is in line with general good programming principles[7]. The contract also contains a constructor function that assigns the value of the input parameter x to variable *num*. The gas-efficient version and the gas-inefficient version are the same except that there is no explicit assignment to variable *num* in the gas-efficient version. According to the Ethereum Yellow Paper [4], the constructor will be executed immediately after the state variables are created during the deployment of the contract to the blockchain. In other words, no matter what value is assigned to variable *num* at the time of declaration, the value will have no effect. Explicit assignment declared in the gas-inefficient version is unnecessary and wastes gas. Moreover, they are conversion methods from bytecode-to-bytecode, which makes the optimization approaches opaque, because the user cannot intuitively know what changes have been made on the contract. And the opaqueness makes users anxious when using the optimized smart contracts because smart con-

```
contract GasInefficiency{
    uint num = 0;
    constructor (uint x){
      num = x;
    }
}
```

(a)

```
contract GasEfficiency{
    uint num; //change
    constructor (uint x){
      num = x;
    }
}
```

(b)

Fig.1. (a) Gas-inefficient contract and (b) its efficient version.

[3] https://solidity.readthedocs.io, Dec. 2021.

[4] https://ethereum.github.io/yellowpaper/paper.pdf, Dec. 2021.

[5] https://ethgasstation.info/, Dec. 2021.

[6] https://etherscan.io/chart/gasused, Dec. 2021.

[7] https://google.github.io/styleguide/cppguide.html#Line_Length, Dec. 2021.

tracts control their assets. Last but not least, these approaches are based on time-consuming satisfiability modulo theories, which will lead to the inability to exhaust the search space and make optimization methods neither sound nor complete.

In this paper, to address the above problem, we firstly collect gas-inefficient patterns of smart contracts by manually going through more than 25 000 posts from Ethereum StackExchange[8] which is the most popular questions and answers (Q&A) website of Ethereum and is recommended by Solidity official[9]. Secondly, in order to make the optimization process transparent, we propose a gas optimization approach utilizing the abstract syntax tree (AST) to optimize the smart contract from the source code level and construct a source code to source code conversion. Finally, we apply our approach in more than 160 000 open-source smart contracts deployed on the Ethereum to demonstrate the universality of the gas-inefficient patterns and economic benefits if the gas-inefficient patterns are removed from the smart contracts. The contributions of this paper are the followings.

- This is the first study that focuses on optimizing the gas consumption of smart contracts at the source code level rather than at the bytecode level. In this way, we can grasp information from source code, optimize from a high level, and thus make up for the shortcomings of existing optimization approaches which have unintuitive optimization results and cannot optimize the smart contract snippets with long opcode sequences.
- We have defined six gas-inefficient patterns from more than 25 000 posts, and conducted a large-scale empirical study on more than 160 000 real smart contracts to evaluate the universality and economic benefits of the patterns.
- To support independent verification or replication of our study, we have provided a replication package[10] in this paper. This package, which contains the gas optimization approach based on AST from source code and the verified smart contracts with gas-inefficient patterns deployed in Ethereum, can be useful for other researchers who are interested in studying the gas optimization of smart contract.

Paper Organization. The rest of the paper is organized as follows. Section 2 introduces the background knowledge of this paper. Section 3 describes the process of defining the gas-inefficient patterns in the smart contract. Section 4 introduces the gas-inefficient patterns and their detection and optimization approach. Section 5 conducts a large-scale empirical study to evaluate the effectiveness and usefulness of the patterns. Section 6 presents the threats to validity. Section 7 discusses related work and Section 8 concludes this paper.

2 Background

Blockchain. Blockchain is a chain data structure that combines a number of blocks in chronological order. It is maintained by all nodes in a decentralized system. Specifically, each node is connected and interacted by P2P networking, and executes, verifies and disseminates the effective transaction data generated within a period of time. At the same time, the node uses the Merkle tree, hash algorithm, timestamp and other technologies to organize transactions into blocks, and compete for the right to append the self-generated block to the chain according to the consensus algorithm. Finally, the node who obtains the right can append the self-generated block at the end of the blockchain and gets corresponding rewards. The remaining nodes update the blockchain. Since the transactions have been validated by all the nodes in this way, the blockchain has the characteristics of decentralization, trustlessness, and tamper-proof.

Ethereum. As a blockchain system with a built-in Turing complete programming language and the first introduction of the concept of smart contracts, Ethereum is currently the most popular smart contract development platform [2]. The core of Ethereum is the Ethereum virtual machine (EVM) which can run smart contracts compiled into the EVM bytecode by the compiler (e.g., solc). It is surprising that roughly 16 000 000 smart contracts have been deployed in Ethereum and these contracts are called more than 1.1 billion times from June 30, 2015 (the launch of Ethereum) to June 30, 2019, suggesting that the smart contract is being used frequently by everyone.

Smart Contract. A smart contract is a program running on a blockchain which needs to be deployed in the blockchain before being called. Specifically, the smart contract creator writes the contract with Solidity, compiles the contract into the EVM bytecode, and sends the transaction containing the bytecode and the parameters of the contract constructor to the nearest Ethereum node. Through the network consensus

[8] https://ethereum.stackexchange.com/, Dec. 2021.

[9] https://docs.soliditylang.org/en/v0.8.9/index.html?highlight=Ethereum%20StackExchange#getting-started, Dec. 2021.

[10] https://github.com/ICSE-Detecting-Gas-Inefficient-Patterns/replication-package, Dec. 2021.

process, the contract is deployed on each blockchain node and the contract address is returned to the user. Anyone can call the smart contract deployed in the blockchain. Specifically, the contract caller sends the transaction containing the contract address, the signature and the input parameters of the called function to the nearest Ethereum node. Since the bytecode is made up of multiple opcodes, running smart contracts is equivalent to performing each opcode in the bytecode (opcode sequence) of contracts.

Gas. In order to avoid the excessive consumption of resources and to ensure termination, Ethereum adopts the gas mechanism charging execution fee from transaction senders. The transaction fee is computed by $GasPrice \times GasUsed$, where the term gas is a special unit to measure the consumption of storage and computing resources. *GasPrice* is constant generally and *GasUsed* by the transaction is stipulated by Ethereum's core protocol. Table 1 shows the amount of gas due to various opcodes. It is noting that the operations require more storage and computing resources cost more gas. For example, the opcode SSTORE requires 5 000 or 20 000 gas which is used to access the storage. When the storage value is set from non-zero to zero, it costs 20 000 gas. Otherwise, it costs 5 000 gas. The opcode ADD requires 3 gas which is used by arithmetic operation.

Table 1. Partial Gas Costs in Ethereum

Operation	Opcode	Gas
Arithmetic operation	ADD/SUB	3
	ADD/SUB	5
	ADDMOD/MULMOD	8
Stack operation	POP	2
	PUSH/DUP/SWAP	3
Unconditional jump	JUMP	8
Get input data of current environment	CALLDATALOAD	3
Memory operation	MLOAD/MSTORE	3
Storage operation	SLOAD	200
	SSTORE	5 000/ 20 000
Get balance of an account	BALANCE	400
Compute Keccak-256 hash	SHA3	30

Note: The complete list can be found in [4].

3 Our Process of Defining Gas-Inefficient Patterns

Smart contracts are programs that run on the blockchain. In order to avoid the infinite loop of contracts, a gas mechanism is introduced. The mechanism stipulates that every time when the smart contract executes an instruction, it consumes a certain amount of gas. Gas optimization of the smart contract can decrease the cost of deploying and calling the contract, which is of great significance. It is noted that informal documentation contained in resources such as Q&A websites (e.g., Ethereum Stack Exchange[11]) is a precious resource for smart contract developers, where the examples of gas-inefficient smart contracts, as well as opinions on how to optimize can be found. Therefore, in this section, we aim at defining a set of new smart contract gas-inefficient patterns from the posts of Ethereum Stack Exchange which is the official discussion site recommended by the Solidity development team. Specifically, we first crawl the posts containing questions, answers and comments in Ethereum Stack Exchange made before July 9, 2020. Second, we utilize keywords to filter out irrelevant posts to improve efficiency. Third, we manually filter irrelevant posts further. Then, we create cards for each post and divide the cards into several themes. The card records the information about the gas-inefficient patterns in the posts. Finally, we subdivide the cards into several categories for each theme. The remainder of this section will introduce the key steps in the process of defining gas-inefficient patterns.

3.1 Filtering out Irrelevant Posts by Keywords

It is time-consuming to find the required information from more than 25 000 posts. Although each post has some tags, it is not feasible to filter posts by these tags, because some posts only contain gas-optimization-unrelated labels but point out the gas-inefficient patterns. For example, the tags of this post[12] are Solidity, contract-development, exceptions and revert-opcode which seem to be irrelated to gas optimization. The post intuitively discusses the use of exception-related Solidity keywords, but one of the answers analyzes the pros and cons of keywords in terms of gas efficiency. Although the labels of the post are not relevant to gas optimization, the post includes gas-inefficient patterns and the analysis of patterns. To solve this problem, we customize some keywords that are closely related to gas optimization, and use these keywords to filter out irrelevant posts instead of using

[11] https://ethereum.stackexchange.com, Dec. 2021.

[12] https://ethereum.stackexchange.com/questions/15166/difference-between-require-and-assert-and-the-difference-between-revert-and-thro, Dec. 2021.

tags given by the site[13]. Specifically, two authors of this paper read Ethereum White Paper [2], Ethereum Yellow Paper [4] and Solidity Developer Documents, and record some keywords related to gas optimization. Then the keywords selected by the two authors are merged and are used to filter irrelevant posts. It is noting that we exploit a two-person collaboration to make the possibility of missing important posts as low as possible. Finally, we utilize 32 keywords to filter out 22 583 irrelevant posts from 25 701 posts. Posts containing less than five keywords will be filtered. In other words, there are 3 118 posts that are likely to be related to gas optimization in Ethereum Stack Exchange.

3.2 Filtering out Irrelevant Posts Manually

Although we use keywords to filter a large number of unrelated posts, there are some posts which contained keywords but are not related to gas optimization. For example, the keyword gas appears many times in this post[14], but this post is just a Q&A about what is gas. To solve this problem, we filter out the posts not related to gas optimization manually. Specifically, for each post filtered by keywords, two authors mentioned in Subsection 3.1 independently judge the relevance between the post and the gas optimization. Then, the opinions of the two authors are summarized. For the posts with different opinions, we invite another author of this paper to judge the relevance of them. Finally, we manually filter out 2 644 irrelevant posts and get 474 related posts for mining gas-inefficient patterns.

3.3 Open and Closed Card Sorting

For the remaining posts by previous filtering steps, two authors of this paper are arranged to create cards for them. One of them creates one or more cards for each post after reading the post carefully, and the other is responsible for checking the cards to avoid missing important information. The card contains the information of gas-inefficient pattern title and description. If the optimization methods are clearly described in the post, the card will contain a solution to the pattern as well.

It is noting that we will not create cards for all gas-inefficient patterns mentioned in the post, because some gas-inefficient patterns have been optimized by the latest version of the compiler (currently the latest version is 0.7.0) and Solidity official recommends that it is best to use a recent version of the Solidity compiler to write smart contracts (the outdated compilers contain some known vulnerabilities[15]). One of the gas-inefficient patterns that have been optimized is the use of the deprecated Solidity keyword *throw* to throw an exception instead of using *require*(), *assert*() and *revert*(). The compiler optimizes this pattern by removing deprecated keywords in version 0.5.0.

We create a total of 532 cards for gas-optimization related posts. After creating the cards, we randomly choose 106 (about 20% of the total) cards and arrange two authors of this paper to classify these cards together according to the reasons for the gas inefficiency. The two authors first carefully read all the contents of the card. Then, they discuss the reason for the inefficiency, and determine a theme for the card. Finally, these 106 cards are divided into the following themes: inappropriate storage, misusing expensive operations, inefficient opcode sequence, and large-scale data storage. The two authors sort together to ensure that they have the same classification criteria for cards. After determining the theme of the card, the two authors independently classify the remaining cards. In the rest of the card classification process, no new categories are found.

3.4 Defining Gas-Inefficient Pattern from Posts

After categorizing the card, we merge similar cards in each theme, summarize the causes of gas-inefficiency more finely and define the gas-inefficient patterns. For inappropriate storage and misusing expensive operations, we define two and four gas-inefficient patterns respectively. For inefficient opcode sequence, we do not define the gas-inefficient pattern because the optimization methods on this theme are not in the source code but in the bytecode, which is not considered in this paper. For the large-scale data storage, the optimization methods of this theme are about reducing the scale of stored data, such as only storing important data in the blockchain. It cannot be solved by changing the source code of a contract. Therefore, the optimization methods of this theme are not within the scope of this paper. The six gas-inefficient patterns which are defined in this subsection will be introduced in Section 4.

[13] https://ethereum.stackexchange.com/tags, Dec. 2021.

[14] https://ethereum.stackexchange.com/questions/3/what-is-meant-by-theterm-gas, Dec. 2021.

[15] https://solidity.readthedocs.io/en/v0.7.0/bugs.html, Dec. 2021.

4 Gas-Inefficient Pattern in Smart Contracts

In Section 3, we define six gas-inefficient patterns from the inappropriate storage theme and the misusing expensive operations theme. Table 2 shows the information of gas-inefficient patterns. Next, we introduce each pattern utilizing Table 2 and the following template.

Description: A description of the gas-inefficient pattern.
Example: An example of the pattern.
Optimization: An optimization of the pattern.

4.1 Inappropriate Storage

4.1.1 Sparse Storage

Description. The storage of the Solidity smart contract is contiguous 32-byte (256-bit) slots. State variables are arranged in slots in the order of definition. For a state variable, if the remaining space of the current slot can store this variable, the variable is stored in this slot. Otherwise the variable is stored in a new (unused) slot. Sparse storage means that variables in the smart contract can be stored in fewer slots. If the definition of state variables is rearranged reasonably, we can use fewer slots to store variables and reduce the cost of using smart contracts. As we all know, the storage resource of the blockchain is set to be very expensive to prevent users from misusing resources. The more the storage a smart contract uses, the more the gas it consumes and the higher the cost is. Therefore, it is meaningful and beneficial to reasonably arrange the state variables (minimizing the wasted space).

Table 2. Gas-Saving Overview of Three Frontier Gas Optimization Studies and Ours

ID	Pattern	Example (Gas-Inefficient)	Example (Gas-Efficient)	Address

1	Sparse storage	`uint8 public decimals;`
		`uint256 public totalSupply;`
		`address public owner;`
			`uint8 public decimals;`
			`address public owner;`
			`uint256 public totalSupply;`
				0x6e56d4e9de4e9d64ccfadce52cbf10c78f096af6

2	Using smaller values for storage without packing	`uint32 public contractVersion`
		`  = 20191203;`
		`string public contractClass`
		`  = "xpetoTimestampLogger";`
		`string public xpectoMandator`
		`  = "xpecto";`
			`uint256 public contractVersion`
			`  = 20191203;`
			`string public contractClass`
			`  = "xpetoTimestampLogger";`
			`string public xpectoMandator`
			`  = "xpecto";`
				0x0a4d60ba4ba53ab7f358ab06976e6b6c4ea132ba

3	Repeat assignment	`contract ProofOfWeakFOMO {`
		`  address owner = msg.sender;`
		`  constructor () {`
		`    owner = msg.sender; }}`
			`contract ProofOfWeakFOMO {`
			`  address owner;`
			`  constructor () {`
			`    owner = msg.sender; }}`
				0xbfd77110f80695f1a97fed29b06abc25dfd447a7

4	Frequent use of state variables	`contract ShareTokenSale {`
		`  uint256 public endTime;`
		`  function startSale() {`
		`    for(...){`
		`      endTime=endTime.add(x); }}}`
			`contract ShareTokenSale {`
			`  uint256 public endTime;`
			`  function startSale() {`
			`    uint256 temp0 = endTime;`
			`    for(...){`
			`      temp0=temp0.add(x); }`
			`    endTime = temp0}}`
				0x13955f1867a0bfbac3146b58ab33b982d72f06e7

5	Without considering the short-circuiting rules	`if((msg.value>=this.balance)`
		`&& (frozen == false)) {`
		`  msg.sender.`
		`  transfer(this.balance); }`
			`if((frozen == false) &&`
			`(msg.value>=this.balance)){`
			`  msg.sender.`
			`  transfer(this.balance); }`
				0xa96e6dbf0f21cfcc9934ad52dec8229e3321254e

6	Inaccurate function visibility	`contract ImmAirDropA {`
		`  function signupUserWhitelist`
		`  (address[] _userlist)`
		`  public{...}}`
			`contract ImmAirDropA {`
			`  function signupUserWhitelist`
			`  (address[] _userlist)`
			`  external{...}}`
				0x015ccd5ad83e95b5cb91b920f689ea329d096190

Note: The Pattern column represents the pattern name; the Example (Gas-Inefficient) column and the Example (Gas-Efficient) column represent the corresponding real examples of gas-inefficient patterns and its efficient versions respectively; the Address column represents the address of the contract where the examples come from.

Example. The contract snippet in the first row and the third column of Table 2 has a gas-inefficient pattern with sparse storage. The snippet has three state variables, called *decimals*, *totalSupply* and *owner*. The variable types of state variables are uint8, uint256 and address, which are occupying 8 bits, 256 bits and 160 bits respectively. According to the variable definition order in contract, the storage of contract is shown in Fig. 2(a). The contract uses three slots to store the variables. We can find that neither variable *decimals* nor variable *owner* occupies the full slot, only a part of the slot. If variable *decimals* and variable *owner* can fit in a single slot, the contract can use two slots only to store state variables instead of three slots, which can optimize the gas usage of the contract and save money.

Optimization. For pattern 1, the optimization is to find the definition order of state variables so that the smart contract uses the least number of slots. Specifically, we first construct a mapping table about the size of variable types. There are two types of smart contract state variables: the reference data type (e.g., struct, array) and the basic variable type (e.g., uint, address). For reference data types, we set their size to 256 bits, because the reference data types always begin in a new storage slot according to the Ethereum Yellow Book [4]. For the basic variable type, we keep their size consistent with the size described in the Solidity developer documentation. Then, we count the number of slots used in the original variable definition order. Finally, we use heuristic rules to adjust the order of variables until we find the order that uses fewer slots than the original. If we can find such an order, then we apply this order to the smart contract to achieve optimization. Otherwise, we judge that there is no sparse storage pattern in the smart contract. To describe our optimization method more specifically, the optimization method is applied to the gas-inefficient example of pattern 1. The method adjusts the original variable order to {*decimals*, *owner*, *totalSupply*}, which is shown in the first row and the fourth column of Table 2. The storage diagram of the optimized contract is shown in Fig.2(b). The method adjusts the definition order of variable *decimals* and variable *owner* to be continuous so that variable *decimals* and variable *owner* can be stored in the same slot and use one less slot. This uses less storage space and consumes less gas, thus saving money.

4.1.2 Using Smaller Values for Storage Without Packing

Description. As mentioned in the previous pattern, the EVM works with 256-bit words. If a variable cannot be packed and stored in the same slot with the surrounding variables, the variable will be stored separately in a 256 bits slot. For smaller values which are less than 256 bits, they will be converted into 256 bits by EVM firstly, and then they are stored in the slot. Compared with storing 256-bit variables in the slot separately, storing variables less than 256 bits in the slot separately will cost extra gas because of conversion.

Example. The contract snippet in the fifth row and the third column of Table 2 is an example of using smaller values for storage without packing. The variable type of the first variable *contractVersion* is uint32 whose size is 32 bits. Since the variable type of the second variables is a string with 256 bits in size and cannot share the slot with other variables, the variable *contractVersion* can only be stored in one slot. Before variable *contractVersion* is stored in the slot, variable *contractVersion* will be expanded to 256 bits by EVM filled with 0. This operation causes additional gas consumption.

Optimization. The optimization method of this pattern is to set the type of the variable that is not packed and less than 256 bits to the 256 bits variable type corresponding to the original variable type (i.e., changing uint8 to uint256, and changing int8 to int256). In addition, for the variables which interact with the variables with changed types, we also need to change their varia-

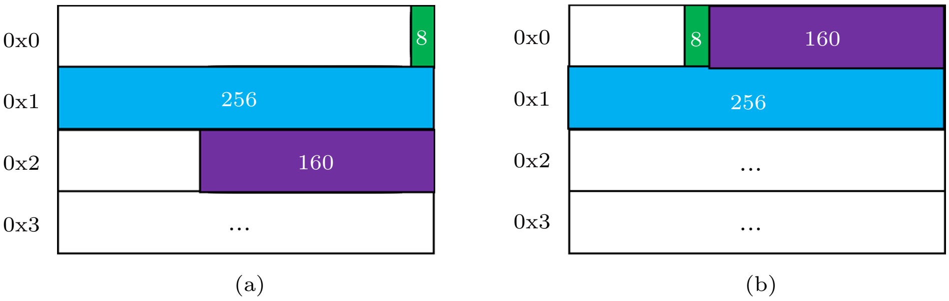


Fig.2. Storage layout. (a) Sparse. (b) Compact.

ble types accordingly. The snippet in the fifth row and the fourth column of Table 2 is an efficient version of the gas-inefficient example of pattern 5. The variable type of variable *contractVersion* is changed to uint256. Since no other variables are assigned by variable *contractVersion*, no more changes to the contract are required.

4.2 Misusing Expensive Operations

4.2.1 Repeated Assignment

Description. According to the Solidity developer document, the compiler will generate some instructions for smart contract deployment. These instructions achieve three tasks: 1) storing the bytecode of the smart contract into the blockchain; 2) storing state variables in the storage of the corresponding contract; if the state variable is not initialized when it is declared, the compiler will automatically initialize the variable to 0; 3) executing the constructor of the smart contract. However, many contract developers intuitively assign values to the contract immediately when the variable is declared due to the programming habits of traditional programming languages (e.g., Java). Therefore, the phenomenon of assigning values to variables in both state variable declarations and constructors frequently occurs in smart contracts. It is noting that this phenomenon belongs to the gas-inefficient pattern because some meaningless code (the code that implements assigning values at state variable declaration) wastes gas. After assigning a value to the state variable during declaration, the state variable is not read or written, and the value is immediately modified in the constructor.

Example. The contract snippet in the second row and the third column of Table 2 is an example of repeated assignment. Smart contract *ProofOfWeakFOM* has a state variable *owner*, and assigns *msg.sender* (representing the sender of the transaction) to the variable when declared. In the constructor, the variable *owner* is modified to *msg.sender*. Obviously, the assignment statement "= *msg.sender*" in variable declaration statement "*address owner = msg.sender*" is meaningless contract snippet, which leads to a waste of gas.

Optimization. The optimization method of this pattern is to remove meaningless contract snippets. Specifically, we scan the contract and record the state variables assigned at the time of declaration and the state variables assigned in the constructor firstly. Then, we find the state variables that are assigned in both the declaration and the constructor by comparing the two variable lists obtained in step 1. Finally, we delete the assignment in the declaration of variables found in step 2. For the gas-inefficient example of pattern 2, the assignment statement "= *msg.sender*" in variable declaration statement "*address owner = msg.sender*" is deleted, which is shown in the second row and the fourth column of Table 2.

4.2.2 Frequent Use of State Variables

Description. According to the Ethereum Yellow Paper, we can know that reading and writing state variables is very expensive compared with reading and writing local variables, because it involves the opcodes SLOAD and SSTORE which consume more gas compared with other opcodes. In particular, if the operation of reading and writing state variables is performed in a loop, the gas consumed by the contract increases with the number of loops.

Example. The contract snippet in the third row and the third column of Table 2 is an example of frequent use of state variables. Smart contract *ShareTokenSale* in example frequently uses state variable *endTimes* in the loop of the *startSale* function. The state variable *endTimes* is read and written once in each loop, which causes EVM to perform a lot of SLOAD and SSTORE opcodes, and the execution of the contract will consume a lot of gas.

Optimization. We notice that although reading and writing state variable is very expensive (it consumes 200 units gas to perform each SLOAD and 20 000/5 000 units gas to perform each SSTORE), reading and writing local variables only involve MLOAD and MSTORE opcodes which consume three units gas when they are performed. Therefore, the optimization method is using local variables to temporarily replace the state variables in loop. Specifically, we find out the state variables used in the loop firstly. Then we declare new local variables to save the value of the state variables and use the local variables to replace the state variables in loop. Finally, we assign the value of the local variables back to state variables. For smart contract *ShareTokenSale* in Table 2, we create a new local variable *temp*0 and assign the value of the state variable *endTimes* to it before the loop is executed. Then, we use variable *temp*0 to replace variable *endTimes* in the loop and assign the value of variable *temp*0 to variable *endTimes* after the loop is executed. So far, the smart contract *ShareTokenSale* has been optimized by reducing the reading and writing of the state variable. The efficient version is shown in the third row and the fourth column of Table 2. Although this optimization will increase the gas consumption of the deployment contract because the amount of code has increased, the optimization effect will become more and more obvious as the number of contract calls increases. It is noting that the contract is deployed only once and it will be called many times. Therefore such optimization is still meaningful.

4.2.3 Without Considering the Short-Circuiting Rules

Description. According to the Solidity developer documentation, the common short-circuiting rules are applicable to the operators $\|$ and &&, which means that in the expression $f(x)\|g(y)$, if $f(x)$ evaluates to true, $g(y)$ will not be evaluated even if it may have side-effects. In other words, the left operand (e.g., $f(x)$) of the logical operators $\|$ and && must be evaluated, while the right operand (e.g., $g(y)$) of the logical operators $\|$ and && has a chance not to be evaluated. Therefore, if a more expensive operation is set as the right operand to try to reduce the number of expensive operations executed, it will save a lot of gas. However, many developers do not consider the short-circuiting rules when developing smart contracts, which leads to implementations with a higher gas consumption.

Example. The contract snippet in the fourth row and the third column of Table 2 is an example of not considering the short-circuiting rules. The snippet has a conditional statement, in which there is a logic operator &&. The logic operator conducts a logic judgment on the balance comparison result ($msg.value \geqslant this.balance$) and the variable comparison result ($frozen \geqslant$ false). Since the operator && applies the short-circuiting rules, if the balance comparison result evaluates to false, the variable comparison result will not be evaluated in the conditional statement. Moreover, balance comparison involves the opcode BALANCE which is a relatively expensive opcode. The balance comparison is more expensive than the variable comparison. Setting the balance comparison operation to the left operand is a waste of gas.

Optimization. The optimization method of this pattern is to place the more expensive operation as the right operand of logical operation. Specifically, for each logical operation in the smart contract, we calculate the gas consumption of the left operand and the right operand of the logical operation firstly. If the gas consumption of the left operand is greater than the gas consumption of the right operand, we exchange the positions of the two operands. Otherwise, it remains unchanged. The snippet in the fourth row and the fourth column of Table 2 is an efficient version of the gas-inefficient example of pattern 4. In the efficient version, the operands on both sides of the logical operation swap positions, and the balance comparison operation becomes the right operand.

4.2.4 Inaccurate Function Visibility

Description. In order to better introduce this pattern, the data location and function visibility of Solidity will be introduced first. There are three data locations (where the data is stored) in Solidity, namely memory, storage and calldata. The calldata is a read-only byte-addressable space where the data parameter of a transaction or call is held. Memory is a volatile read-write byte-addressable space. It is mainly used to store data during execution, mostly for passing arguments to internal functions. Storage is a persistent read-write word-addressable space. This is where each contract stores its persistent information. There are four kinds of function visibility, namely public, external, internal and private. This pattern mainly focuses on public and external. Public function can access by all. External function cannot be accessed internally, which means that it can only be accessed by transaction. The visibility of external is a subset of the visibility of public. If the visibility of the function is not specified, it will be defaulted to public according to the Solidity developer documentation. We notice that because the developer did not actively set the function visibility and did not know the keyword external which is not a common keyword in other programming languages, the visibility of many contracts can be set to external but was set to public, which causes extra gas consumption. However, public functions consume more gas than external functions because a public function needs to copy all of the function input arguments to memory from calldata to enable the public function to be called internally. Memory allocation (i.e., MSTORE) is expensive. Changing the visibility of the function which is public and is not invoked by other functions of the same contract to external will save gas.

Example. The contract snippet in the sixth row and the third column of Table 2 is an example of inaccurately setting the visibility of functions. The visibility of function *signupUserWhitelist* in the snippet is public and the function is not called by other functions in the contract.

Optimization. The optimization method of this pattern is to set the visibility of public functions that are not called internally to external. The snippet in the sixth row and the fourth column of Table 2 is an efficient version of the gas-inefficient example of pattern 6. The visibility of function *signupUserWhitelist* is changed to external.

5 Results of Empirical Study

Here we conduct a large-scale empirical study to see how generalizable our characterized gas-inefficient patterns are and how much will be saved if the gas-inefficient patterns are removed from the smart contracts.

5.1 Dataset

In order to evaluate the universality and usefulness of the gas-inefficient patterns proposed in this paper, we conduct experiments on more than 160 000 smart contracts, which is the largest empirical study of open-source smart contracts as far as we know. However, it is non-trivial to collect such a large number of open-source smart contracts and their related information. Specifically, from Etherscan[16], we crawl 160 301 open-source smart contracts which have been deployed on Ethereum and have been open sourced in Etherscan before June 19, 2020. The reason why there are so many open-source contracts in Etherscan is that Etherscan has gathered many contract users by providing an easy-to-use Ethereum block explorer and contract developers attract users to use their contracts by opening the source code of contracts on Etherscan. It should be noted that the difficulty in collecting open-source smart contracts is that Etherscan only shows the last 500 open-source contracts[17]. To solve this problem, we have recorded open-source contracts every once in a while. Hence, we can get more than 160 000 open-source contracts in Etherscan. Moreover, in the process of crawling the source code of the smart contracts, we not only collect the source code of the contracts, but also collect information such as the creation time of the contract and the address of the creator.

5.2 Experimental Environment and Tools

All our experiments are performed on a machine with 16 GB memory and 1 TB disk space, equipped with an Intel® Core™ i5-4690K CPU @ 3.50 GHz and running 64-bit Ubuntu 20.04.1 LTS operating system. In the process of detecting and optimizing smart contracts, we traverse the AST utilizing solidity-parser-antlr tool[18].

To closely simulate the actual contract deployment and usage, we firstly prepare 74 compilers with their versions ranging from 0.1.1 to 0.6.11, thereby we can deploy the contract using the same compiler as that used by the running instance on the Ethereum network. After compiling the contract, we use Truffle framework v4.1.13[19] to automatically deploy the contract in the test chain Ganache v1.1.0[20] to obtain the gas consumption of deploying contracts. The bytecodes of function arguments are firstly decoded into a human-readable form using an argument decoder based on the canoe-solidity library v0.1.0[21] according to the ABI Specification[22] before being passed into Truffle.

5.3 Prevalence Analysis

We use the approach proposed in this paper to detect the gas-inefficient patterns on 160 301 open-source smart contracts deployed on Ethereum. The detection result is shown in Fig.3. The x-axis represents the serial number of the pattern and the y-axis represents the total number of smart contracts with the corresponding pattern. Pattern 2 is the most common pattern among these six patterns. They account for 45.79% of all contracts. It indicates that many contract developers do not fully consider the use of variable types, only consider the storage space required by the variable, and ignore the gas consumption caused by the variable type. Pattern 4 is the rarest pattern among these six patterns. They only account for 2.72% of all contracts. It indicates that developers will pay attention to avoiding reading and writing state variables in the loop to save gas. Moreover, 52.75% (84 566/160 301) of contracts contain at least one gas-inefficient pattern, which illustrates that about half of the contracts can be optimized. The number of contracts that only contain 1, 2, 3, 4, 5, and 6 gas-inefficient patterns is 38 263, 34 454, 8 964, 2 356, 426 and 103 respectively. It indicates that most

[16] https://etherscan.io/, Dec. 2021.

[17] https://etherscan.io/contractsVerified, Dec. 2021.

[18] https://www.npmjs.com/package/solidity-parser-antlr, Dec. 2021.

[19] https://www.trufflesuite.com/, Dec. 2021.

[20] https://www.trufflesuite.com/ganache, Dec. 2021.

[21] https://github.com/cryptofinlabs/canoe-solidity, Dec. 2021.

[22] https://solidity.readthedocs.io/en/v0.7.0/abi-spec.html, Dec. 2021.

optimizable contracts contain one or two gas-inefficient pattern(s). There are 41 245 optimizable contracts that contain both pattern 2 (P2) and pattern 5 (P5), which means that pattern 2 and pattern 5 often appear together.

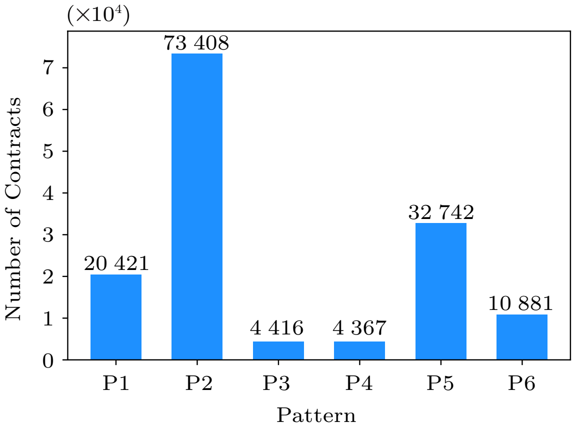


Fig.3. Prevalence of six gas-inefficient patterns.

5.4 Money Saved Analysis

To illustrate the economic benefit of removing the gas-inefficient patterns proposed in this paper from the smart contracts, we compare gas consumption for each contract before and after optimization. We acquire more real gas consumption by replaying the original transactions of the contract to reproduce the contract execution. Specifically, we develop a gas consumption comparison program with a contract deployer module and a transaction replayer module, which is shown in Fig.4, where the notation of an arrow indicates data flow.

The deployer module firstly compiles and deploys the contract using a compiler with exactly the same version and configurations as that of the running instance on the real Ethereum network to obtain the same bytecode, thereby the gas consumptions we get are more realistic. When the contract is deployed, the replayer

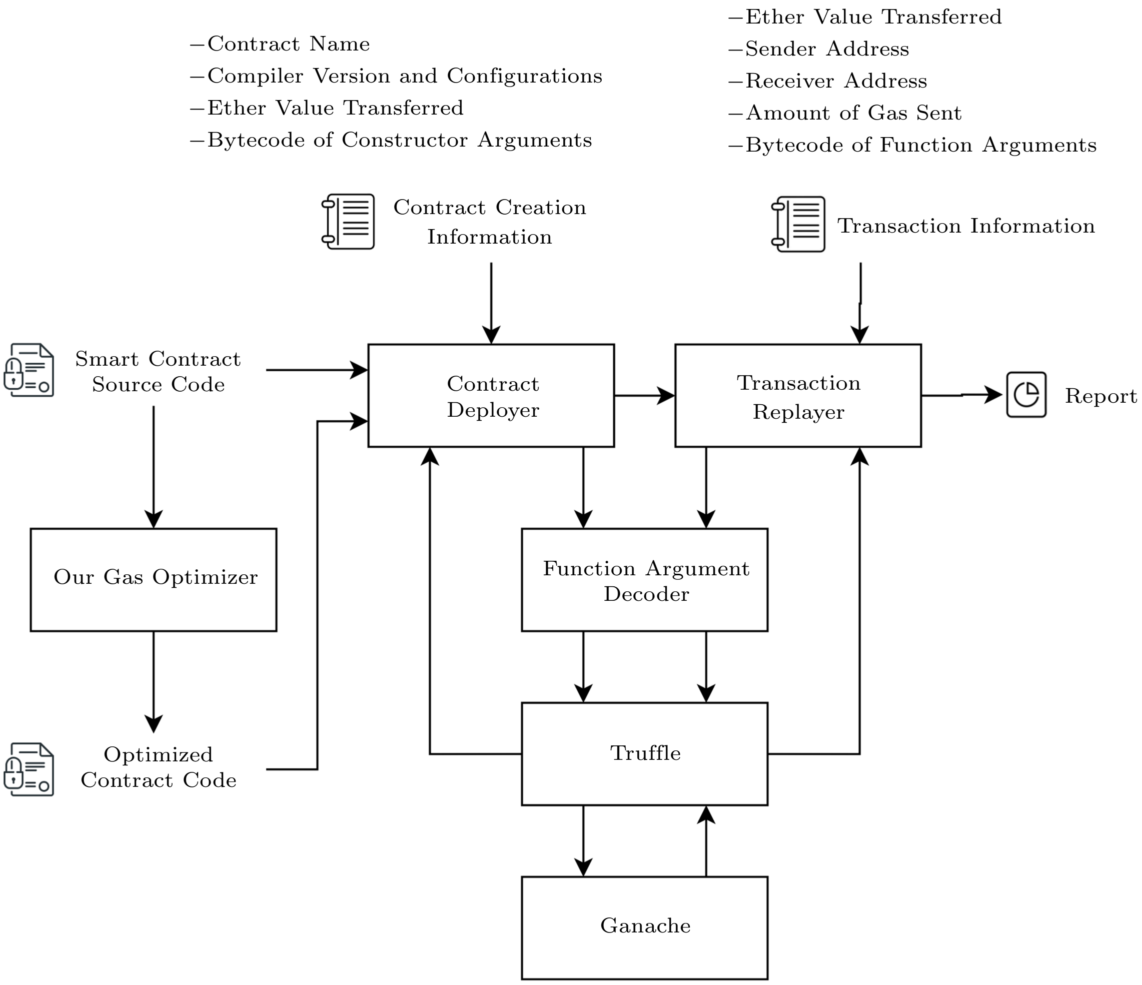


Fig.4. Architecture of gas consumption comparison.

module utilizes transaction information recorded on the real Ethereum network and replays them accordingly. It is noting that since it is unfeasible to have the same sender addresses on our test chain as on the Ethereum network, we make a mapping from the real addresses to the addresses on our test chain, and replace the sender addresses in each transaction with our mapped addresses before replaying it. Finally, the program generated a report describing the gas consumption of each of the contracts covered by the six patterns, before and after optimization.

Due to time constraints, during the experiment, we simply ignore contracts with more than 1 000 transactions and contracts containing transactions from more than 100 different sender addresses. Also, we only try to deploy at most 20 000 contracts of each pattern and to replay their transactions. The contracts which fail to compile, revert during the deployment process, or encounter EVM internal errors during the replaying process are not included in the result. Eventually, 44.2% of our tests were successful. In total, the experiments take 364 hours on our machine.

The gas saving of each pattern is shown in Table 3. From the perspective of contract development, pattern 6 is the best economic benefit pattern among these six gas-inefficient patterns. Since the external function lacks the operation of copying the input parameters to the contract memory compared with the public function, the opcode sequence of the optimized contract is much shorter than that of the original one and pattern 6 can significantly save gas during deployment. In the same way, the worst economic benefit is pattern 4 when deploying contracts, because the optimized contracts have added a local variable declaration and two assignment operations. The opcode sequence of the optimized contract becomes longer. From the perspective of contract invoking, pattern 4 is the best economic benefit pattern. The optimized contract uses local variables to reduce the reading and writing of state variables which are relatively expensive operations in Ethereum, reducing the execution of expensive operations, thereby saving a lot of gas. In general, considering that gas price is about 1.15×10^{-7} Ether and one Ether can be exchanged into around $380 in August 2020, at least $0.30 can be saved per contract. Since there are more than 16 million smart contracts deployed in Ethereum, if gas optimization is performed before deploying contracts, it will bring huge economic benefits.

Table 3. Gas Saving of Six Gas-Inefficient Patterns

Pattern	#c	#t	#Saving Gas	
			Deploy	Invoke
P1	11 657	286 696	80 494 110	12 174 283
P2	8 810	229 995	79 077 573	749 724
P3	2 149	59 146	12 262 199	8 860 627
P4	1 898	69 621	−22 030 761	23 219 821
P5	7 442	189 270	348 897	4 850 887
P6	3 216	109 215	29 664 663	23 636 855

Note: The #c column and the #t column represent the number of contracts and invoking contract transactions corresponding to the pattern involving the experiment respectively. The #Saving Gas (deploy) column and the #Saving Gas (invoke) column represent the gas saved when deploying the contract and invoking the contract after optimization respectively.

5.5 Comparison with Existing Approaches

Table 4 is an gas-saving overview of three recent frontier gas optimization studies and our study. The #contract row and the #transaction row represent the number of contracts and invoking contract transactions in the experiment respectively. The #saving gas (total) row represents the amount of gas saved by deploying and invoking contracts after optimizing. The #saving gas (avg) row represents the average gas saved per interaction with the contracts, that is, $\#savinggas(avg) = \#savinggas(total)/(\#contract + \#transaction)$. Since these three tools are not all open source, we cannot conduct an experiment applying their approaches and our approaches to the same dataset. The data of these three advanced gas optimization approaches come from their papers. Since Albert *et al.*[3] did not describe the number of transactions in the paper, rows 3 and 5 of the fourth column are empty. Chen *et al.*[5] were the first to study the problem of smart contract gas optimization. They defined 24 gas-inefficient patterns and acquired the saved gas by replaying all transactions in Ethereum as of June 10, 2017. Their average gas saved is 1 524.73. With the development of Ethereum, the number of contracts and transactions in Ethereum is growing rapidly. Replaying all transactions to acquire gas consumption is very time-consuming and resource-intensive. Therefore, the studies of [6] and [3] optimized the top 1 500 and 2 500 contracts that are most frequently invoked in Ethereum respectively rather than all contracts in Ethereum. GasChecker[6] optimizes contracts by removing three gas-inefficient patterns and Albert *et al.*[3] optimized contracts with logical reasoning. The average gas saved of GasChecker[6] was 21.69. These three gas optimization studies focus on optimizing the contracts at the bytecode level while we focus on the source code

Table 4. Gas-Saving Overview of Three Frontier Gas Optimization Studies and Ours

	#Contract	#Transaction	#Saving Gas (Total)	#Saving Gas (Avg.)
Chen *et al.* [5]	386 906	5 663 971	9 225 940 756	1 524.73
GasChecker [6]	1 500	7 000 000	151 899 834	21.69
Albert *et al.* [3]	2 500	—	1 309 875	—
Ours	25 278	679 830	172 495 693	244.64

level. Similarly, we only optimize partial contracts. We define six gas-inefficient patterns and our average gas saved is 244.64. It is noting that we only calculate saved gas for contracts that contain less than 1 000 transactions. These transactions may not invoke the optimized snippets, thereby the actual average gas saved will be larger.

6 Threats to Validity

Internal Validity. Some gas-inefficient patterns may be missed because custom keywords are not comprehensive enough and the authors responsible for extracting the information of gas-inefficient patterns from posts may not be careful enough. In order to deal with this problem, we arrange two persons to customize keywords and synthesize their keywords. We arrange two persons to extract information from the posts, one is responsible for the extraction, and the other is responsible for checking whether there are any omissions. In addition, in order to evaluate the patterns proposed in this paper more realistically, we acquire the gas consumption by replaying all transactions of contracts. However, there is a slight discrepancy between the reproduction situation and the real situation. One reason is that we cannot obtain all versions used by the contract creators because some compilers are experimental compilers which have only been open to the public for a short time. Another reason is that the operation of the smart contracts utilizes some external environment elements (e.g., calling other smart contracts). To reduce this threat, we use a compiler whose version is as consistent as possible with the version of the compiler used by the contract creator. And we optimize smart contracts that do not involve external elements.

External Validity. We conduct our study on more than 160 000 real smart contracts. We find that our uncovered patterns and the corresponding problematic and justifiable cases are common among the studied smart contracts. However, our finding may not be generalizable to other smart contract programming languages that are not based on EVM because the cost of these programming languages for resource consumption is very different from that of EVM-based languages.

7 Related Work

There are three areas of research related to this paper: 1) empirical studies on smart contracts, 2) optimizing the gas consumption of smart contracts, and 3) anti-patterns.

7.1 Empirical Studies on Smart Contracts

Despite the fact that Ethereum and smart contracts are relatively new, many empirical studies have been performed on smart contracts. Oliva *et al.* [7] conducted a study of smart contracts deployed in Ethereum from the inception of Ethereum (July 30, 2015) until September 2018. They focused on three aspects: activity level, category, and code complexity. Durieux *et al.* [8] presented an empirical evaluation of automated analysis tools on 47 587 Ethereum smart contracts. Chen *et al.* [9] conducted an empirical study to understand and characterize smart contract defects. They defined 20 kinds of contract defects from the posts of Ethereum Stack Exchange. Two of the defects are related to gas-inefficiency. The first is to use variable type byte[]. They recommended using bytes instead. After verification, there is no difference in gas consumption between these two variable types in the latest version of the compiler. The second is pattern 6 introduced in this paper. Our work is different from them because we are fully concerned about the gas-inefficiency of smart contracts and we conduct a qualitative and quantitative analysis of the proposed gas-inefficient patterns on more than 160 000 real contracts. As far as we know, it is the largest empirical study of open source smart contracts.

7.2 Optimizing the Gas Consumption of Smart Contracts

Smart contracts have been explored for a variety of different contexts, including vulnerability detection [10–12], testing [13, 14] and decompilation [15].

The most relevant work for this research is to optimize the gas consumption of smart contracts.

The EVM bytecode superoptimizer ebso[4] optimizes the bytecode of smart contracts by encoding the operational semantics of EVM instructions as SMT formulas and leveraging a constraint solver to automatically find cheaper bytecode. Its experiments indicated that it hit a time-out in nearly 82% of all test contracts. Albert *et al.*[3] improved ebso by combining symbolic execution with an effective Max-SMT encoding. Max-SMT encoding improves the speed of optimization for each smart contract to avoid the timeout problem of ebso.

Gasper[16] defines seven gas-inefficient patterns. GasChecker[6] adds four new patterns and removes one pattern that never appeared on the basis of Gasper. GasChecker can also detect these 10 patterns with symbolic execution and optimizes three of the patterns. Chen *et al.*[5] identified 24 gas-inefficient patterns by manually scrutinizing the execution traces of real under-optimized smart contracts. These patterns are composed of 1–5 opcodes. They also built a tool to detect and optimize the patterns. The above studies detected and optimized the contract at the bytecode level, while our study is at the source code level. The gas-inefficient patterns defined in this paper have no intersection with the above studies, except for pattern 4. For pattern 4, GasChecker[6] can detect it and the approach proposed in this paper can detect and optimize it.

In addition to the above related work focused on bytecode, there are some related studies focused on source code. SmartCheck[17] translates Solidity source code into an XML-based intermediate representation and checks gas-inefficient patterns against XPath patterns. It could check gas-inefficient patterns about byte arrays and costly loop. SolidityCheck[18] also detects these two gas-inefficient patterns from the source code level. But it uses regular expressions to define the characteristics of patterns and uses regular matching the patterns. Correas *et al.*[19] proposed a static profiling technique for contracts with static resource analysis. It could automatically detect and optimize gas-expensive fragments where state variables were frequently used where in loops.

7.3 Anti-Patterns

Anti-pattern detection and elimination are applied in various aspects of software engineering involving logging code[20], continuous integration (CI)[21], IOS app[22] and database application[23]. Li *et al.*[20] manually studied over 3k duplicate logging statements and their surrounding code in four large-scale open-source systems: Hadoop, CloudStack, ElasticSearch, and Cassandra. They uncovered five patterns of duplicate logging anti-patterns which may affect developers' understanding of the dynamic view of the system and developed an automated static analysis tool to detect these patterns. CI-Odor[21] is a tool for detecting four anti-patterns that reduce the promised benefits of CI by analyzing regular build logs and repository information. Afjehei *et al.*[22] manually studied 225 performance issues collected from four open source iOS apps and uncovered four performance anti-patterns. They also implemented a static analysis tool to help developers avoid these performance anti-patterns in the code. SQLCheck[23] is a holistic toolchain for automatically finding and fixing anti-patterns in database applications. It can rank the anti-patterns based on their impact on performance, maintainability, and accuracy of applications, and changes to the database design to fix these anti-patterns.

8 Conclusions

Snippets with gas-inefficiency will cause unnecessary costs to smart contract users. In this paper, we defined six gas-inefficient patterns from more than 25 000 posts. Different from the existing studies, we proposed a detection and optimization approach of gas-inefficient patterns at the source code level, which is more transparent and considers inefficient programming patterns caused by developers. Further, we applied the detection and optimization approach to a dataset containing more than 160 000 real smart contracts. Experimental results demonstrated that there are 52.75% of contracts with at least one gas-inefficient pattern proposed in this paper, and each contract can reduce at least \$0.30 in fees if the gas-inefficient patterns are removed from the contract. Considering that there are more than 16 million smart contracts in Ethereum, if gas optimization is performed before contract deployment, it will save a lot of money. Considering the limitations of manual analysis, future studies will use the natural language processing technology to assist manual analysis to reduce artificially introduced deviations. In order to discover more gas-inefficient patterns, future work will mine gas-inefficient patterns from commit messages which contain meaningful information about code changes.

References

[1] Zheng Z, Xie S, Dai H N, Chen W, Chen X, Weng J, Imran M. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 2020, 105: 475-491. DOI: 10.1016/j.future.2019.12.019.

[2] Zheng P, Zheng Z, Wu J, Dai H N. XBlock-ETH: Extracting and exploring blockchain data from Ethereum. *IEEE Open Journal of the Computer Society*, 2020, 1: 95-106. DOI: 10.1109/OJCS.2020.2990458.

[3] Albert E, Gordillo P, Rubio A, Schett M A. Synthesis of super-optimized smart contracts using Max-SMT. In *Proc. the 32nd International Conference on Computer Aided Verification*, Jul. 2020, pp.177-200. DOI: 10.1007/978-3-030-53288-8_10.

[4] Nagele J, Schett M A. Blockchain superoptimizer. arXiv:2005.05912, 2020. https://arxiv.org/abs/2005.05912, May 2021.

[5] Chen T, Li Z, Zhou H, Chen J, Luo X, Li X, Zhang X. Towards saving money in using smart contracts. In *Proc. the 40th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results*, May 27-Jun. 3, 2018, pp.81-84. DOI: 10.1145/3183399.3183420.

[6] Chen T, Feng Y, Li Z, Zhou H, Luo X, Li X, Xiao X, Chen J, Zhang X. GasChecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 2020, 9(3): 1433-1448. DOI: 10.1109/TETC.2020.2979019.

[7] Oliva G A, Hassan A E, Jiang Z M. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering*, 2020, 25(3): 1864-1904. DOI: 10.1007/s10664-019-09796-5.

[8] Durieux T, Ferreira J F, Abreu R, Cruz P. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proc. the 42nd IEEE/ACM International Conference on Software Engineering*, Oct. 2020, pp.530-541. DOI: 10.1145/3377811.3380364.

[9] Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T. Defining smart contract defects on Ethereum. *IEEE Transactions on Software Engineering*, 2022, 48(1): 327-345. DOI: 10.1109/TSE.2020.2989002.

[10] Jiang B, Liu Y, Chan W. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proc. the 33rd IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2018, pp.259-269. DOI: 10.1145/3238147.3238177.

[11] Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2018, 2(OOPSLA): Article No. 116. DOI: 10.1145/3276486.

[12] Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B. ReGuard: Finding reentrancy bugs in smart contracts. In *Proc. the 40th IEEE/ACM International Conference on Software Engineering: Companion*, May 27-June 3, 2018, pp.65-68. DOI: 10.1145/3183440.3183495.

[13] Li Z, Wu H, Xu J, Wang X, Zhang L, Chen Z. MuSC: A tool for mutation testing of Ethereum smart contract. In *Proc. the 34th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2019, pp.1198-1201. DOI: 10.1109/ASE.2019.00136.

[14] Wang X, Wu H, Sun W, Zhao Y. Towards generating cost-effective test-suite for Ethereum smart contract. In *Proc. the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, Feb. 2019, pp.549-553. DOI: 10.1109/SANER.2019.8668020.

[15] Grech N, Brent L, Scholz B, Smaragdakis Y. Gigahorse: Thorough, declarative decompilation of smart contracts. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.1176-1186. DOI: 10.1109/ICSE.2019.00120.

[16] Chen T, Li X, Luo X, Zhang X. Under-optimized smart contracts devour your money. In *Proc. the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*, Feb. 2017, pp.442-446. DOI: 10.1109/SANER.2017.7884650.

[17] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. SmartCheck: Static analysis of Ethereum smart contracts. In *Proc. the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, May 27-June 3, 2018, pp.9-16. DOI: 10.1145/3194113.3194115.

[18] Zhang P, Xiao F, Luo X. SolidityCheck: Quickly detecting smart contract problems through regular expressions. arXiv:1911.09425, 2019. https://arxiv.org/abs/1911.09425, Nov. 2021.

[19] Correas J, Gordillo P, Román-Díez G. Static profiling and optimization of Ethereum smart contracts using resource analysis. *IEEE Access*, 2021, 9: 25495-25507. DOI: 10.1109/ACCESS.2021.3057565.

[20] Li Z, Chen T H, Yang J, Shang W. DLFinder: Characterizing and detecting duplicate logging code smells. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.152-163. DOI: 10.1109/ICSE.2019.00032.

[21] Vassallo C, Proksch S, Gall H C, Di Penta M. Automated reporting of anti-patterns and decay in continuous integration. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.105-115. DOI: 10.1109/ICSE.2019.00028.

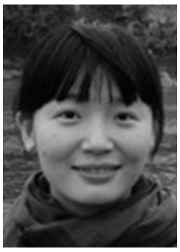
[22] Afjehei S S, Chen T H, Tsantalis N. iPerfDetector: Characterizing and detecting performance antipatterns in iOS applications. *Empirical Software Engineering*, 2019, 24(6): 3484-3513. DOI: 10.1007/s10664-019-09703-y.

[23] Dintyala P, Narechania A, Arulraj J. SQLCheck: Automated detection and diagnosis of SQL anti-patterns. In *Proc. the 2020 ACM SIGMOD International Conference on Management of Data*, Jun. 2020, pp.2331-2345. DOI: 10.1145/3318464.3389754.

Que-Ping Kong received her M.E. degree in computer science and engineering in Sun Yat-sen University (SYSU), Guangzhou, 2018. She is currently working towards his Ph.D. degree in SYSU, Guangzhou. Her current research interests include smart contract, program analysis and software engineering.

Zi-Yan Wang received his B.E. degree in computer science in South China Normal University, Guangzhou, 2019. He is currently a postgraduate student at the Sun Yat-sen University, Guangzhou. His research interests include programming languages and software engineering.

Yuan Huang is an associate professor at the Sun Yat-sen University, Zhuhai. He received his Ph.D. degree in computer science from Sun Yat-sen University, Zhuhai, in 2017. He is particularly interested in software evolution and maintenance, code analysis and comprehension, and mining software repositories.



Xiang-Ping Chen is an associate professor at the Sun Yat-sen University, Guangzhou. She got her Ph.D. degree in computer software and theory from the Peking University, Beijing, in 2010. Her research interests include software engineering and mining software repositories.

Xiao-Cong Zhou received his Ph.D. degree in computer software and theory from Institute of Software, Chinese Academy of Sciences, Beijing, in 2001. Now he is an associate professor at Sun Yat-sen University, Guangzhou. His research interests include empirical software engineering, code analysis and comprehension, and software testing.

Zi-Bin Zheng received his Ph.D. degree from the Chinese University of Hong Kong, Hong Kong, in 2011. He is currently a professor at School of Computer Science and Engineering with Sun Yat-sen University, Guangzhou. He published over 300 international journal and conference papers, including nine ESI highly-cited papers. His research interests include blockchain, artificial intelligence, and software reliability. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE 2010, and the Best Student Paper Award at ICWS 2010.

Gang Huang is a full professor at Peking University Shenzhen Graduate School, Shenzhen. He received his Ph.D. degree in computer software and theory from Peking University, Beijing, in 2003. His current research interests include operating systems, cloud computing, and Internetware.