GaSaver: A Static Analysis Tool for Saving Gas

Ziyi Zhao, Jiliang Li, Zhou Su, and Yuyi Wang

Abstract—Smart contracts are programs running on Ethereum, whose deployment and use require gas. Gas measures the cost of performing specific operations as an index designed to quantify the computing power consumption. Existing unoptimized smart contracts make contract developers and users spend extra gas. To save gas and optimize smart contracts, this paper proposes a new tool named GaSaver for automatically detecting gas-expensive patterns based on Solidity source code. Specifically, we first identify 12 gas-expensive patterns in smart contracts and classify them into three categories: storage-related, judgment-related, and loop-related. Then, we deploy gas-expensive patterns and group them into three levels according to gas waste degree. By conducting extensive experiments on real data sets, we find that 89.68% of the 1172 smart contracts suffer from gas-expensive patterns, 94.27% of 1100 new smart contracts are gas-expensive, and 80.56% of 72 widely used smart contracts are affected. Finally, the experiment results show that the proposed GaSaver can effectively optimize smart contracts. Besides, the proportion of gas-expensive cases in widely used smart contracts is lower than that in the newly released smart contracts.

Index Terms—Smart contract, gas-expensive pattern, code optimization, static analysis, solidity

1 Introduction

Bitcoin [1] has gained increasing interest as the first cryptocurrency since its emergence in 2008. Its underlying technology, blockchain, has shown various application scenarios. The functionality of blockchain technology has gone from peer-to-peer bitcoin payments in the beginning to being used in an increasing number of other areas. As Bitcoin's script is not Turing-complete [2], [3], it only allows users to process transactions automatically by coding the script, limiting it to being used only for cryptocurrency transactions. To expand its use, Buterin [4] identified the applicability of decentralized computing beyond transactions and designed Ethereum. Ethereum supports executing programs written in the Turing-complete language, enabling users to develop and use decentralized applications (DAPPs) in Ethereum and apply blockchain technology to different areas [5], such as banking, insurance, and arts.

The computer programs that automatically enforce the contract terms are smart contracts (SCs) [6]. SCs allow users to conduct unchangeable transactions without a third party under mutual distrust through the consensus mechanism of blockchain. SCs can be written in high-level languages (the most mature development language is Solidity) and then compiled into the form of Ethereum Virtual Machine (EVM) bytecode. SCs are deployed onto the blockchain through broadcasting EVM bytecode and initialization parameters, and are represented by a unique 16-bit address. Currently, Ethereum is one of the most popular blockchains, with over 1 million transactions in a single day, the number of new SCs added to Ethereum is around 500 per day [7].

Although the current SCs have a broad application in Ethereum, some factors still hinder its further use. For instance, gas waste is a typical issue [8]. Gas is a unit of measurement in Ethereum, an index designed to quantify computing power consumption. With the calculated units of gas, we can easily calculate how much gas developers need to consume to deploy a SC, how much gas users need to spend on completing a transaction, and how much block rewards miners can receive when they complete the packaging confirmation a block. The introduction of gas mechanism can ensure the termination of SCs and prevent resource abuse [9]. When SC runs, the peer's computing resources are consumed during operating. In Ethereum, every EVM operation needs to consume a certain amount of gas. When Ethereum initiates each transaction, the transaction initiator presets a certain amount of gas limit. The transaction will fail if the gas runs out during execution [10].

Actually, some SCs with expensive operations can be completed via lower gas operations. We call these SCs as *unoptimized SCs*, and operations that consume more gas are called as *gas-expensive patterns*. Compared with optimized SCs, unoptimized SCs cost more transaction gas [11]. For example, lots of gas is consumed if variables in storage are repeatedly called from the source code level. Gas-expensive patterns bring redundant gas consumption to both developers and users. Specifically, gas is the cost incurred by the computational resources consumed by each opcode during the SC execution. The reduction of gas represents a reduction in the number of opcodes that need to be executed [12].

However, SCs are based on blockchain technology. As a distributed transaction book that maintains the same ledger

- *Ziyi Zhao is with the School of Software Engineering, Xi'an Jiaotong University, Xi'an 710049, China. E-mail: ziyi.zhao@stu.xjtu.edu.cn.*
- *Jiliang Li and Zhou Su are with the School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an 710049, China. E-mail: jiliangli@foxmail.com, zhousu@ieee.org.*
- *Yuyi Wang is with ETH Zürich, 8092 Zürich, Switzerland, and also with CRRC Zhuzhou institute, Zhuzhou, Hunan 412007, China. E-mail: yuwang@ethz.ch.*

Manuscript received 14 June 2022; revised 26 September 2022; accepted 7 November 2022. Date of publication 11 November 2022; date of current version 8 June 2023.
This work wass supported in part by National Key R&D Program of China under Grant 2022YFB2702800 and in part by NSFC under Grants U20A20175, U1808207, and 62102305.
(Corresponding author: Jiliang Li.)
Recommended for acceptance by D. Zeng.
Digital Object Identifier no. 10.1109/TSUSC.2022.3221444

among untrustworthy multiple parties, the blockchain is tamper-proof because it consists of a chain structure with timestamps [13]. So even if the gas-expensive pattern is detected, the SC cannot be modified [14]. To save gas, developers should replace the gas-expensive code with optimized code before deploying SCs onto Ethereum, which not only reduces the capital consumption of developers and users but also reduces the consumption of electricity and contributes to the achievement of carbon neutrality [15].

There has been some research on the detection of gas-expensive patterns, such as GASPER [11] and GasChecker [16]. GASPER is a tool for finding gas-costly patterns by detecting the EVM bytecode of SCs. GasChecker is a tool to find ten gas-inefficient patterns based on symbolic execution. Most previous works primarily focus on different gas-expensive patterns and suggest only practical tools for developers. Our research focuses on improving coding practices that may negatively impact gas consumption. Previous works on loop-related patterns were not comprehensive and did not verify whether these patterns increased gas consumption.

To our best knowledge, most previous detect tools are bytecode based [11], [16], [17], [18], [19], while source-code oriented detection is more friendly. A survey showed that 63.4% respondents think gas optimization is miserable, especially for complex applications. One problem that makes gas optimization difficult is the lack of gas estimation tools at the source code level [12]. Developers usually prefer typing and optimizing source code rather than bytecode since working at the source code level is more intuitive [12]. Therefore, to optimize the source code on gas consumption, developers have to turn to the existing gas estimation tools at the bytecode level (such as Remix [20]), which cannot fully reflect the effect of source code changes. Most developers mentioned the high demand for effective gas estimation tools at the source code level [12]. Exploiting tools to optimize smart contract source code rather than bytecode will be crucial. Such tools can directly identify the source code part with the higher gas cost, which will be greatly valued.

To resolve the above issues, we identify 12 gas-expensive patterns and divide them into three categories: storage-related patterns, judgment-related patterns, and loop-related patterns. Our patterns are based on the source code level to help developers better understand which operations waste gas. For each pattern, deterministic scenarios are given with examples of where this pattern can exist. It can allow developers to directly determine the part of the source code with the higher gas cost. We provide developers with an order of magnitude classification criteria by how much gas is wasted to evaluate better the parts of smart contract code that need optimization from a gas consumption perspective. We deploy and test the gas-expensive patterns and classify them into three levels according to gas waste degree: 1-high consumption, 2-medium consumption, and 3-low consumption.

We propose and develop `GaSaver`, a new tool that automatically discovers gas-expensive patterns in Solidity source code. Our data is extracted from the source code of SCs, which is higher than bytecode. By applying `GaSaver`, we select 1172 SCs with verified source code for analysis, of which 1100 new SCs and 72 widely used SCs. We define the new SCs as the newly released SCs, and the widely used SCs as the SCs with higher usage amounts and higher total balances, which are the Ether (ETH) balance in SC account for more than 0.01% of the total amount of Ethereum, whereby ETH is a virtual token in Ethereum. We have tested 1172 SCs and 89.68% of them suffer from gas-expensive patterns, where 94.27% of 1100 new SCs and 80.56% of 72 widely used SCs have gas-expensive patterns, respectively. Hence, the new SCs are not very good at avoiding gas-expensive patterns compared to the widely used SCs, indicating that gas waste is still widespread and has not been taken seriously by many developers.

To summarize, we make the following contributions.

- We summarize 12 gas-expensive patterns and divide them into three categories. We supplement the section of Expensive operations in subsection 3.3.3. To facilitate developers to grasp the gas-expensive patterns better, we listed various situations of each pattern in detail in the introduction of patterns.
- We classify the gas-expensive patterns into three levels according to gas waste degree. This can provide developers with specific order-of-magnitude grading criteria to better determine whether optimizations are necessary.
- We develop `GaSaver`, a new tool for automatically finding gas-expensive patterns in the Solidity source code. After developers code smart contracts, the gas-expensive patterns can be easily aware at the source code level.
- We apply `GaSaver` to 1172 SCs and find that 89.93% of selected SCs suffer from gas-expensive patterns. In addition, to explore and understand whether SC developers will learn and reduce the use of the gas-expensive patterns, we compare the gas-expensive patterns of the new SCs with those of widely used SCs and find that the developers did not pay attention to the optimization upon gas fees.

2 Background

We briefly introduce our working background in this section, including blockchain platform, Ethereum, smart contract, and gas mechanism.

2.1 Blockchain and Ethereum

Blockchain is functionally a decentralized distributed transaction book. It is characterized by tamper-proof, traceability, information sharing, and transparency. People can think of blockchain as a growing list of blocks [21]. Each block is a container data structure that aggregates transaction information, consisting of a block header and a long list of transactions that constitute the main body of the block. Because each block contains the previous block's hash value, the last block cannot be changed or rejected without rejecting all subsequent blocks.

Ethereum is a programmable, visual and easy-to-operate blockchain, which is Turing-complete. It allows users to create complex operations according to their wishes, such as programming SCs and issuing tokens [22]. Ethereum database is maintained and updated by many peers connected to the network, recording all the situations of Ethereum together. Each network peer runs a stack-based EVM and

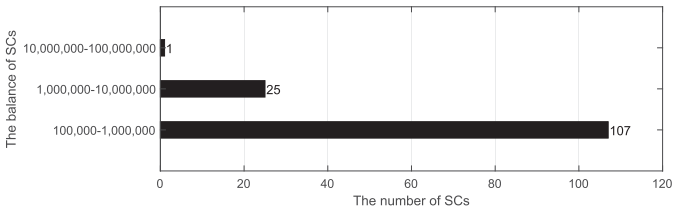


Fig. 1. Distribution of smart contract balances (ether).

executes the same instructions, which are open, transparent, and immutable. EVM is a low-level stack machine with an instruction set [23], [24], including arithmetic operation instructions, comparison operation instructions, cryptographic calculation instructions, stack, memory, storage operation instructions, block, smart contract-related instructions, etc. Data is either stored in the blockchain storage in a permanent data structure or stored in local memory.

2.2 Smart Contract

The smart contract is a computer protocol running on Ethereum designed to disseminate, validate or execute contracts in an informative manner. SCs are usually programmed in high-level languages such as Solidity, Vyper, and Serpent, and then compiled into EVM bytecode and uploaded to the blockchain. After successful uploading, a unique 16-bit address represents a SC. The SC cannot be revoked or deleted [25]. Each SC is associated with predefined executable code and has a certain amount of virtual currency as well as its private storage.

The SC allows trusted transactions without a third party, which can be tracked and irreversible. EVM is used to run SCs, and the execution of SCs depends on their codes and is called by transactions [25], which can come from Externally Owned Accounts (EOA) or other contract accounts [26]. Contract Accounts contain executable bytecode. It cannot initiate transactions actively, but can only be executed according to pre-programming smart contract code after being triggered, while EOA does not contain bytecode, and it can start transactions.

SCs have a significant difference in the balance. According to the data of SCs whose balance accounts for more than 0.01% of the total Ethereum currency on March 20, 2022, the difference in contract balance is very significant (see Fig. 1). The balance of the Eth2 Deposit Contract [27] with the most considerable balance is 10624290.000069 Ether, accounting for 8.85% of the total Ethereum currency.

2.3 Gas Mechanism

Gas [5] is a count that exists inside the EVM and is used to measure the cost of performing specific operations on Ethereum. On the one hand, gas rewards miners for successfully packing blocks as the main incentive mechanism in Ethereum. On the other hand, it can prevent malicious attacks, especially Denial of Service (DoS) attacks, and maintain the normal operation of the Ethereum network. Gas is the fuel in Ethereum, which ensures the operation of Ethereum ecology.

$$G = gas\_cost \times gas\_price \tag{1}$$

Here, G denotes the gas used for performing operations, gas_cost denotes the amount of gas consumed by all performing operations, and gas_price denotes the price of a unit of gas, which is determined by the relationship between market supply and demand. Fig. 2 shows the average gas price of Ethereum in recent years, taken from the website of Etherscan.io [7]. For example, the average gas price on March 10, 2022, was 36.484913763 gwei, equivalent to $3.6484913763 \times 10^{-8}$ Ether, roughly equivalent to 9.5×10^{-5} dollars. Moreover, Ethereum defines a gas limit for each transaction, mainly used to control the upper limit of gas consumption for a single transaction, and the transaction initiator specifies the size. If the gas consumption of instruction execution is bigger than gas limit, the transaction request fails, and the gas cannot be refunded [26].

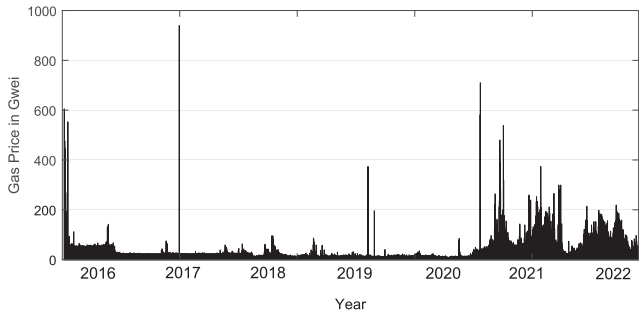


Fig. 2. Ethereum average gas price.

Both deployment and execution of SCs require gas consumption. The opcode is generated when the SC Solidity code is compiled. Each opcode has a predetermined amount of gas consumption, defined by the Ethereum core protocol [26]. Table 1 shows the payable expenses (gas) for some of the operations in Ethereum, details of which are fully tabulated in the Ethereum core protocol, particularly in Appendix G [26].

It is worth noting that SCs consist of many opcodes. Some gas fees of opcodes are lower, such as ADD, AND, and PUSH, because they are pure stack operations. And MSTORE is also cheap to save a word to memory. The "memory" mentioned in Ethereum is temporary data storage, which can only take effect in the current function. Moreover, the gas consumption multiplies if reading or writing more than one word in memory. Some gas fees of opcodes are higher, such as SSTORE is used to update storage, and CREATE is used to create a new account. The "storage" mentioned in Ethereum is a persistent memory area called across functions.

3 GAS-Expensive Patterns

In this chapter, we introduce a series of SC gas-expensive patterns. We identified 12 gas-expensive patterns, divided into three categories: storage-related patterns, judgment-related patterns, and loop-related patterns. The storage-related pattern is a pattern that increases gas costs with memory and storage. The judgment-related pattern is a pattern that improperly sets the conditional statement, and the loop-related pattern is a pattern in which loop operations can be simplified.

3.1 Category 1: Storage-Related Patterns

3.1.1 Use Byte[]

$byte[]$ is similar to $bytes$, but when an external function is called as a parameter, $bytes$ is compressed, which saves

TABLE 1
Gas Costs in Ethereum

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
AND/OR/XOR	3	Bitwise logic operation
POP	2	Stack operation
PUSH/DUP/SWAP	3	
JUMP	8	Unconditional Jump
SLOAD	200	Storage operation
SSTORE	5,000/20,000	
BALANCE	400	Get balance of an account
CALL	700	Paid for a Call operation
CREATE	32,000	Create a new account

more space. Fig. 3 (Pattern 1) gives an example of use $byte[]$ where Line 1 use $byte[]$ type. Therefore, using $bytes$ rather than $byte[]$ in SCs is beneficial.

3.1.2 Initialize Variables With Default Values

In Solidity, all variables have default values, and their byte representations are all zero (0, $false$, $0x0$, and so on, depending on the data type). Therefore, if the variable's default value is zero, there is no need to initialize the variable. Initializing it with default values is an operation waste gas. For example, Fig. 3 (Pattern 2) shows an $uint256$ type variable that should not be initialized with 0.

3.1.3 Use String or Bytes

The size of $string$ or $byte$ variable is dynamic. If the string is short enough, we should use $byte32$. Fig. 3 (Pattern 3) gives an example of use $string$ variable. Any fixed-size variability is cheaper than variable-size variability.

3.2 Category 2: Judgment-Related Patterns

3.2.1 Use Long Reason String

In Solidarity, we can append the required statement to the error reason string to make it easier to understand why the SC calls the restore. Each reason string occupies at least 32 bytes, which will occupy the space in the deployment bytecode. Therefore, to avoid consuming more gas, it is necessary to ensure that the string is within 32 bytes. For example, Fig. 4 (Pattern 4) shows a long error reason string of more than 32 bytes.

3.2.2 Redundancy Checks Caused by SafeMath Library

It is not necessary to check the same conditions in different forms. The $SafeMath$ library causes the most common redundancy check. The $SafeMath$ library will check arithmetic overflow, so there is no need to check whether the variable overflows in the code. In the following code, Fig. 4

Pattern 1
```
1  byte[] someVariable11 ;
```

Pattern 2
```
1  uint256 someVariable12 = 0 ;
```

Pattern 3
```
1  string someVariable13 = "usd bytes32 not string" ;
```

Fig. 3. Storage-related patterns cases.

Pattern 4
```
1  require (b >= a , "Sorry to inform you that this place is
   wrong because balance (b) is less than the amount you
   want to transfer (a). " ) ;
```

Pattern 5
```
1  using SafeMath for uint256;
2    function p5 ( uint256 a , uint256 b ) {
3      require ( b > a ) ;
4      b = b . sub ( a ) ;
5    }
```

Pattern 6
```
1  function p6 ( uint256 a , uint256 b ) {
2    uint[] x = new uint[] ( 100 ) ;
3    require ( a > 1 ) ;
4    require ( b == x . length ) ;
   }
```

Pattern 7
```
1  function p7 ( uint x ) {
2    if ( x < 1 )
3      if ( x > 2 )
4        XXX
5  }
```

Pattern 8
```
1  function p8 ( uint x ) {
2    if ( x <= 0 )
3      if ( x < 1 )
4        XXX
5  }
```

Fig. 4. Judgment-related patterns cases.

(Pattern 5), "b" and "a" are already compared in the $SafeMath$ library when the sub-function is called, so there is no need to compare them again.

3.2.3 Use Multiple Require

The short circuit rule uses $logical\ OR$ ($||$) and $logical\ AND$ ($\&\&$) to sort conditional judgment statements with different costs. If the first expression is true in $logical\ OR$, the second expression cannot be executed. We can put the low-gas-costing operations ahead and the high-gas-costing operations behind. If the previous low-gas-costing operations are feasible, we can skip the later high-gas-costing operations, thus saving gas. Similarly, if the first expression is false in $logical\ AND$, the following expression cannot be evaluated, and sorting can reduce gas consumption. Fig. 4 (Pattern 6) shows two require statements, which we can use the short circuit rule.

3.2.4 Dead Code in Nested Conditional Statements

Dead code is the code that never executes, such as code whose execution conditions can never be met. Like the Solidity code block in Fig. 4 (Pattern 7), we can see two contradictory conditional judgments. If "$x < 1$" holds, "$x > 2$" cannot hold, and if "$x < 1$" does not hold, "$x > 2$" cannot be checked, so the code never executes, which would consume Ethereum gas resources, but has no effect. This kind of pattern usually appears in nested conditional statements.

Pattern 9

```
1  function p9 ( uint x ) {
2    uint m ;
3    uint n ;
4    for ( uint i = 0 ; i < x ; i ++ )
5      m += i ;
6    for ( uint j = 0 ; j < x ; j ++ )
7      n += j ;
8  }
```

Fig. 5. Loop condition repeat cases.

3.2.5 Use Unnecessary Conditional Judgment

The results of some conditional judgments can be understood without code execution, so such conditional judgments can be removed. For example, Fig. 4 (Pattern 8) shows the value range of "$x < 1$" is smaller than "$x <= 0$", "$x < 1$" is true, "$x <= 0$" must be true, "$x < 1$" is not true, and "$x <= 0$" must not be true, so "$x < 1$" is wasting gas.

3.3 Category 3: Loop-Related Patterns

3.3.1 Loop Condition Repeat

If two loops have the same judgment conditions, we can combine multiple loops into one loop like Fig. 5 (Pattern 9), where both "m" and "n" are updated and cannot affect each other, thus reducing the amount of program operation.

3.3.2 Repeat Calculation

If an expression in a loop produces the same result at each iteration, we can move it out of the loop to evaluate it first, then reuse the calculated result to save the extra gas cost in the loop.

This rule contains multiple patterns:

- *Repeat calculation in loop body*: In a loop statement, repeated calculations exist in the loop body, especially some expressions involving expensive operations. Fig. 6 (Pattern 10-1) shows "$a * b$" at Line 6 can be calculated first, and in the loop body, we can reuse the calculated result;
- *Repeat calculation in loop judgement condition*: In a loop statement, repeated calculation exist in the loop judgment condition. Fig. 6 (Pattern 10-2) shows "$a * b$" at Line 4 can be calculated first. Instead of use Repeat calculation in loop condition.

3.3.3 Expensive Operations

The expensive operation of performing gas in a loop is worth noting because it may be called one or more times in a loop, wasting a lot of gas.

This rule contains multiple patterns:

- *Call storage variable in loop judgment condition*: In a loop statement, the storage variable is called in the loop judgment condition. Because of the high cost of SLOAD and SSTORE opcodes, the gas cost of managing storage variables is much higher than that of memory variables, so avoiding manipulating storage variables in a loop is necessary. For example, in Fig. 7 (Pattern 11-1), the "num" at Line 4 is a storage variable, where storage needs to be accessed before each loop.
- *Call the storage variable in loop body*: In a loop statement, the storage variable is called in the loop body. Fig. 7 (Pattern 11-2) shows the "num" at Line 4 is a storage variable, every time the loop is called needs access to more storage.
- *Call expensive operations in loop judgment condition*: Because of the high cost of operations such as fetching array length and finding balance, the expensive operations bring the higher gas cost in loop judgment conditions. For example, in Fig. 7 (Pattern 11-3), the "$.length$" at Line 3 is an expensive operation. Thus, the user's every call consumes extra gas.
- *Call expensive operations in loop body*: Expensive operations in the loop body also bring higher gas costs. Fig. 7 (Pattern 11-4) shows the "$.length$" at Line 4 is an expensive operation, "$x.length$" does not change in the loop, so "$x.length-1$" also is a constant value, we do not need to call it again and again to increase gas consumption.

Pattern 10-1

```
1  uint a = 1 ;
2  uint b = 2 ;
3  function p10 ( uint x ) {
4    uint sum = 0 ;
5    for ( uint i = 0 ; i <= x ; i ++ )
6      sum = sum + a * b ;
7  }
```

Pattern 10-2

```
1  uint a = 1 ;
2  uint b = 2 ;
3  function p10 ( uint x ) {
4    for ( uint i = 0 ; i <= a * b ; i ++ )
5      XXX
6  }
```

Fig. 6. Repeat calculation cases.

3.3.4 One-Side Conditional Execution

If a conditional judgment statement is in the loop, it must be judged whether it is satisfied in each loop iteration. So if the results of each comparison are the same, gas will be wasted. For example, in Fig. 8 (Pattern 12), the "$x > 0$" at Line 2 is a conditional judgment. In this loop, "x" does not change, so this condition has the same result, which can be moved out of the loop to reduce gas.

4 GAS-Expensive Pattern Level

We use `Remix` [20] to compare the source code of the sample gas-expensive patterns with the optimized code under Solidity (V0.7.0), which consumes less gas. Our sample dataset is posted at https://github.com/ziyizhao10/gas-expensive-patterns-sample. We grade the gas-expensive patterns according to the difference in gas consumption before and

Pattern 11-1

```
1  uint x = 0 ;
2  uint num = 10 ;
3  function p11() {
4    for ( uint i = 0 ; i <= num ; i ++ )
5      XXX
6  }
```

Pattern 11-2

```
1  uint num = 0 ;
2  function p11() {
3    for ( uint i = 0 ; i < x ; i ++ ) {
4      num += 1 ;
5    }
6  }
```

Pattern 11-3

```
1  uint[] x = new uint[] ( 100 ) ;
2  function p11() {
3    for ( uint i = 0 ; i < x.length ; i ++ )
4      XXX
5  }
```

Pattern 11-4

```
1  uint[] x = new uint[] ( 100 ) ;
2  function p11() {
3    for ( uint i = 0 ; i < 100 ; i ++ )
4      x[i] = x[x.length - 1] ;
5  }
```

Fig. 7. Expensive operations cases.

after optimization, divided into three levels: 1-high consumption, 2-medium consumption, and 3-low consumption. Table 2 shows gas consumption and pattern levels.

The loop-related patterns use one hundred iterations to calculate the execution cost in the test process. In Table 2, some patterns consume more gas after optimization, but only the gas consumption of calling the loop one hundred times. If more iterations are called, the less the gas cost of the optimized code is consumed, and the more pronounced the effect of saving gas is.

To be sure, with the increase in the number of iterations, the gas-expensive patterns related to loops consume more gas, and the optimized code can achieve the purpose of saving gas. We also test for this problem, for example, the repeated calculation pattern in the judgment condition. After comparison, if the loop is executed one thousand times, the deployment cost increases by 1284 gas, but the execution cost decreases by 207987 gas. It can save 206703 gas. Therefore, we conclude that the more loops the optimized code execute, the more gas consumption can be reduced.

5 Evaluation

5.1 GaSaver

We develop GaSaver, a static analysis tool that automatically discovers gas-expensive patterns from the Solidity source code of SCs. GaSaver can process source code directly without conversion to bytecode. Since our tool is primarily for SC developers, source code detection is more intuitive than detection after conversion to EVM bytecode.

Pattern 12

```
1  for ( uint i = 0 ; i < 100; i++) {
2    if ( x > 0)
3      y += x ;
4    XXX
5  }
```

Fig. 8. One-side conditional execution cases.

GaSaver uses ANTRL [28] for lexical and syntactical analysis of Solidity source code. Fig. 9 shows the flow of lexical parsing and syntactical parsing. In the lexical analysis stage, the character sequence is mainly converted into Token, and in the syntactical analysis stage, the input words construct an Abstract Syntax Tree (AST). Parsers usually use a lexer to separate Token from the input character stream and take the Token as their input. When using ANTRL, for lexical parsing and parsing to work properly, it is necessary to define Solidity grammar, which is the meta-language of ANTRL.

We use ANTRL and Solidity syntax to generate an XML parsing tree as an intermediate representation (IR). We detect gas-expensive patterns by using an XPath query on the IR. The Solidity source code is completely converted to IR, and all its elements are accessible through XPath matching. Thus, GaSaver provides complete code coverage.

As an example, consider the case of calling the storage variable in loop body in expensive operations, such as:

```
1:  for(uint i = 0 ; i < x ; i++) {
2:   num += 1;
3:  }
```

This structure has the parse tree in Fig. 10, and the corresponding XPath pattern is as Listing 1, where "*ancestor*" is to find the ancestor nodes of the current node.

Listing 1. Check the code calling the storage variable in loop body

```
1: //forStatement/statement//expression[expression
   [1]/primaryExpression[identifier=ancestor::
2:   contractDefinition/contractPartDefinition
3:       /(structDefinition/variableDeclaration|
     stateVariableDeclaration)
4:       [typeName/elementaryTypeName[text()[1]
     = "uint"]]
5:   /identifier]]
```

In this case, we don't have a false positive situation because we can accurately describe the structure of the XPath. However, XPath cannot accurately describe some patterns that produce false positives and require manual inspection and analysis. For example, the expensive operation "*.length*" in loop condition has false positive situations as below.

```
1:  for ( uint i = 0 ; i < x.length ; i ++ ) {
2:   XXX
3:  }
```

TABLE 2
Gas-Expensive Pattern Level

Pattern		deploy cost	execution cost	sum	Level
Use byte[]		0		0	3
Initialize variables with default values		2258		2258	2
Use string or bytes		3436		3436	2
Use long reason string		32046	0	32046	1
Redundancy checks caused by SafeMath library	Redundancy checks by require	2364	23	2387	2
	Redundancy checks by if	1728	26	1754	
Use multiple require		192	-8	184	3
Dead code in nested conditional statements		1926	8	1934	2
Use unnecessary conditional judgment		1944	25	1969	2
Loop condition repeat		4962	12600	17562	1
Repeat calculation	Repeat calculation in loop body	-1284	20787	19503	1
	Repeat calculation in loop judgement condition	-21610	13016	-8594	
Expensive operations	Call storage variable in loop judgment condition	22515	12100	34514	1
	Call the storage variable in loop body	-852	21673	20821	
	Call expensive operations in loop judgment condition	-1080	10787	9707	
	Call expensive operations in loop body	-1080	10679	9599	
One-side conditional execution		0	2560	2560	2

The corresponding XPath pattern is as Listing 2, where we do not know whether the "$.length$" changes in the loop.

Listing 2. Check the code calling expensive operations in loop judgment condition

```
1:  //forStatement[(condition | expression[2])
2:    //expression[matches(text()[1], "^\.length
    $")]]
```

5.2 Experimental Results

We initially assume that the new SCs might have more gas-expensive issues, and test new SCs and widely used SCs to verify this assumption in an experiment. From January 4 to January 14, 2022, we selected 100 newly released SCs from Etherscan [7] as new SCs every day, and selected SCs whose balance account for more than 0.01% of the total amount of Ethereum as widely used SCs for experiments.

`GaSaver`, like `Oyente` [29], `SmartCheck` [30] and other tools, cannot detect a false negative (FN) for each item in the code. After `GaSaver` automatically detected the Solidity source code, we manually marked all the problems found as true positive (TP) or false positive (FP) and calculated their false discovery rate (FDR). FDR is the ratio of false positive (FP) among all the findings of the tool

$$FDR = FP/(TP + FP). \tag{2}$$

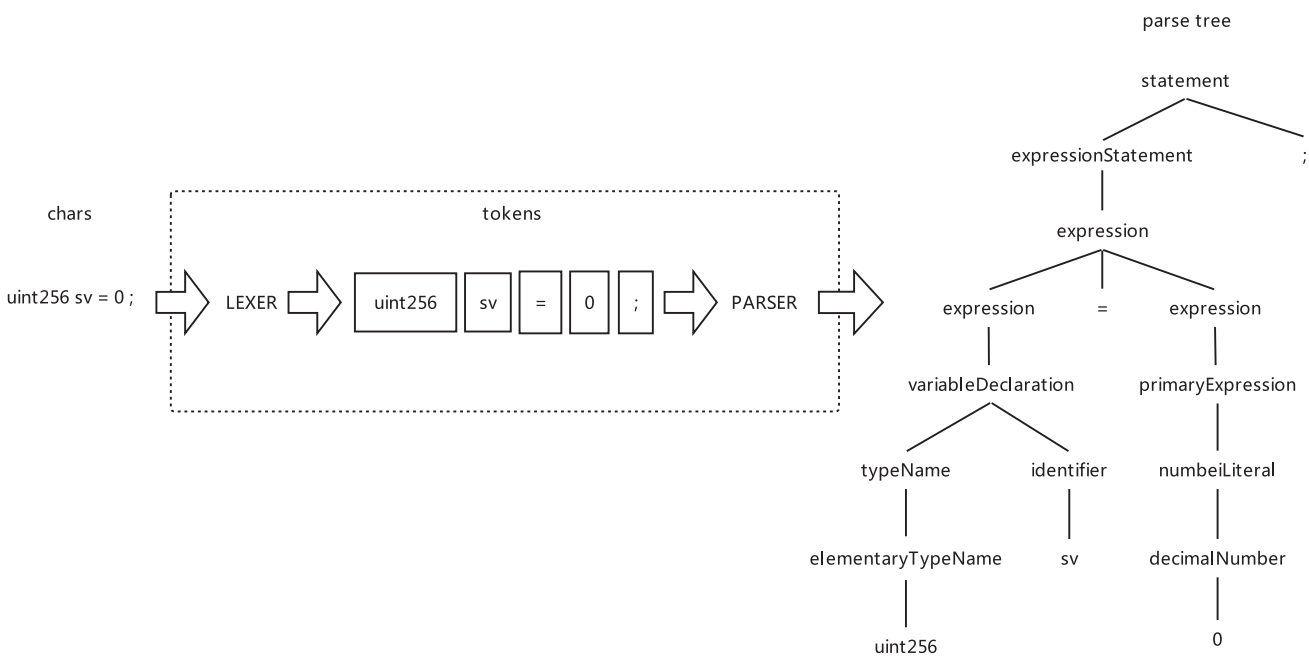


Fig. 9. ANTRL lexical analysis and grammatical analysis process.

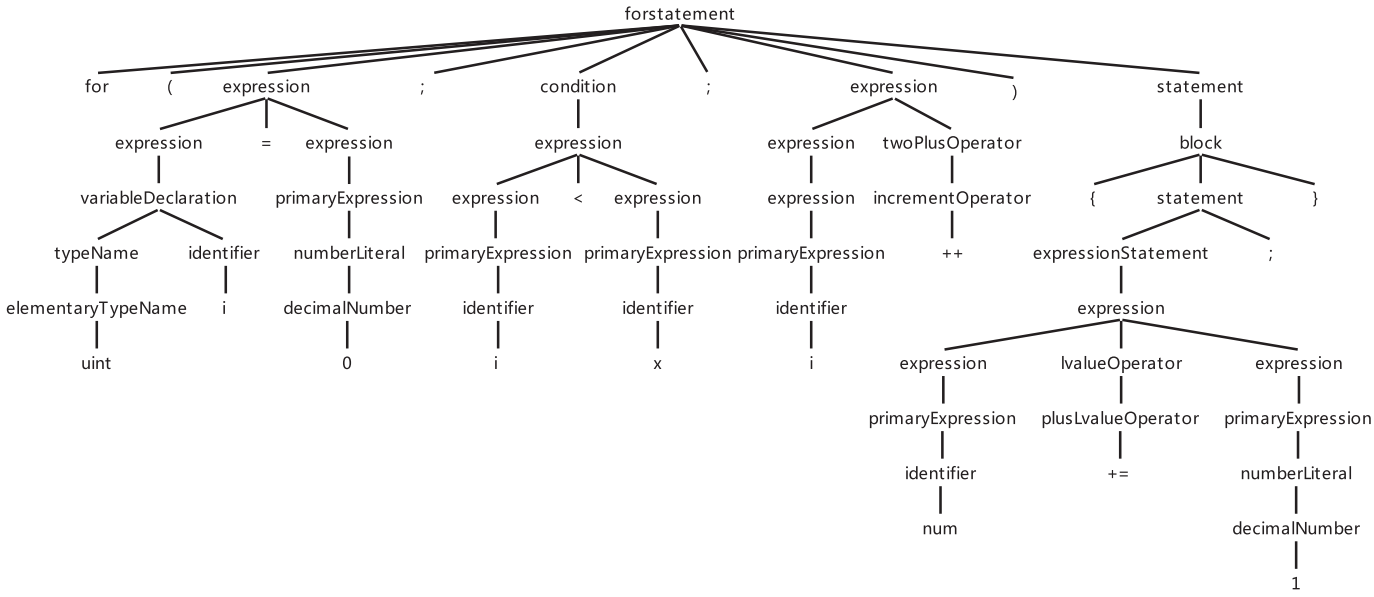


Fig. 10. Parse tree for the case calling the storage variable in loop body in expensive operation.

GaSaver discover that 89.68% of SCs waste gas, and 83.02% of SCs have high consumption gas-expensive patterns. In the analysis of 1100 new SCs and 72 widely used SCs, we find 23371 gas-expensive cases. The research results are shown in Fig. 11 (color indicates severity: yellow-high consumption, red-medium consumption, blue-low consumption). It can be seen that unlike more than 90% of SCs obtained by GASPER [11]. The number of dead code and opaque predicates detected by GaSaver is small, related to the detection method we adopt because these patterns in SCs are related to business logic, which cannot be automatically detected. GASPER is a dynamic detection that can grasp which code blocks are not executed, while GaSaver cannot detect which code blocks are not executed by the static detection method, so the number of detection is much less. Use byte[] pattern in subsection 3.1.1 has not been detected, indicating that byte[] type is rarely used in SCs, or developers have begun to pay attention to this problem.

To compare the gas waste situation between new SCs and widely used SCs, we sort them separately, in which 94.27% of new SCs have gas-expensive problems, and 80.56% of widely used SCs have gas-expensive problems. We list the proportion of gas-expensive patterns in the two SCs in Fig. 12. We can see that widely used SCs have a lower proportion than new SCs, indicating that the existence of gas waste in SCs is still common and has not been taken seriously by many SC developers.

5.3 Case Studies

Real case 1: BearStaking.

BearStaking is deployed at the address $0xEDA3f43bdBfDbb9CEb7646E7EDAF0F0A899B6596$. GaSaver discovers expensive operations in loop at 750 lines, as shown in

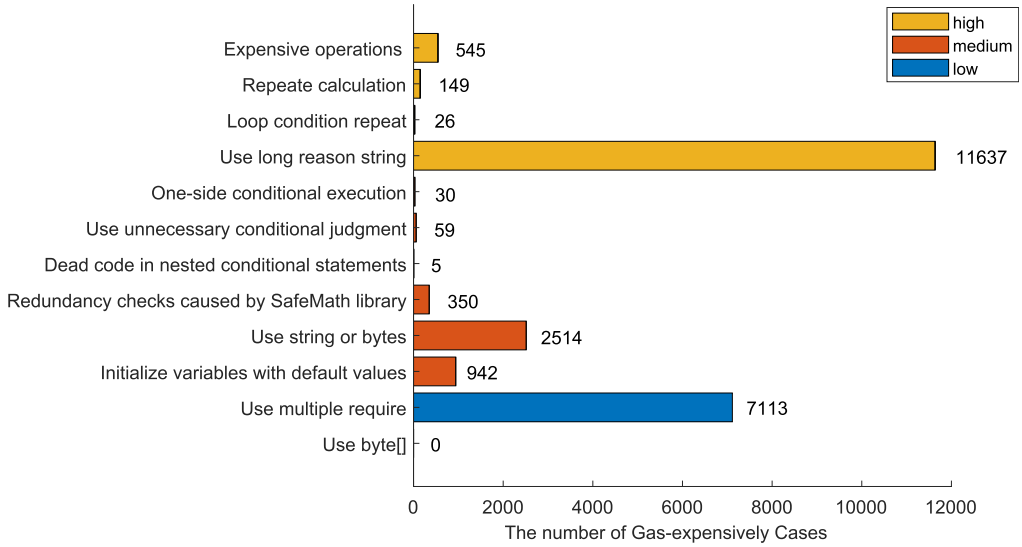


Fig. 11. Detection found gas-expensive patterns in SCs (color indicates severity: yellow-high consumption, red-medium consumption, blue-low consumption).

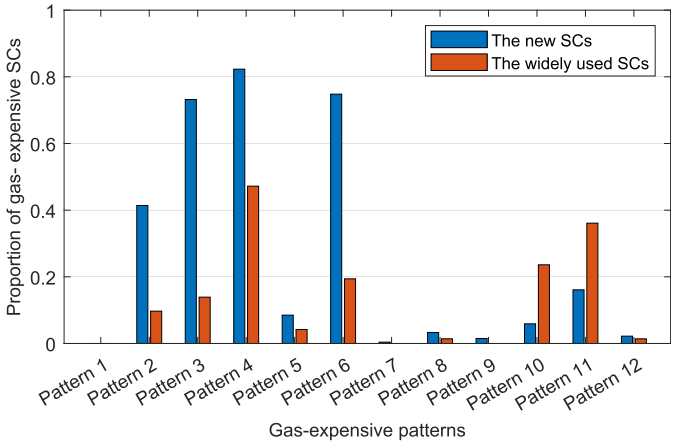


Fig. 12. Proportion of gas-expensive problems in new SCs and widely used SCs.

Fig. 13. Because the "$bears$" array is defined in storage (line 635), the operation of taking the length of bears at line 750 is expensive. We can assign "$bears.length$" as a stack variable and then use the stack variable as a loop judgment condition instead of "$bears.length$" to perform a "$.length$" operation again and again with more gas consumption.

Although gas is also consumed each time the stack variable is called, it is less than the amount of gas consumed in each call to "$bears.length$". Because the calling of "$bears.length$" requires not only calling $bears$ array but also calculating the length of array, assigning "$bears.length$" to a stack variable simplifies the operation. Meanwhile, the loop only calls the variable value each time rather than updating it and the length of the $bears$ array does not change, so the variable will not bring extra gas consumption operations.

Real case 2: Watchdog.

Watchdog is deployed at the address $0xE0509B4DEbF29D12eF5D1C5D858D1331D1125D51$. GaSaver finds a long reason string in line 953, as shown in Fig. 13. We should minimize the reason string to reduce gas consumption.

Real case 3: HGoldStaking 90Contract

HGoldStaking90Contract is deployed at the address $0x7169267D76101676951A8Df9C74289A0880bfA90$. GaSaver discovers in line 157 that repeated caculation is used, as shown in Fig. 13. We should calculate "$2 + 2 * 20$" first and assign its value to the stack variable instead of calculating it each time, resulting in more gas consumption than calling the stack variable.

Real case 4: PuffinPirates.

PuffinPirates is deployed at the address $0xAaCE7c90a03f164ae8da301C64B78E5930dc97f1$. GaSaver finds initializing the bool type variable with the default value false in lines 536, 537, as shown in Fig. 13. It is redundant operation that wastes gas.

Real case 5: TemplateCrowdsale.

TemplateCrowdsale is deployed at the address $0x5784B41db244E8AF2d55eEeFC07281BeDb0c4587$. GaSaver discovers a redundancy check on the SafeMath library at line 834, as shown in Fig. 13. When the SafeMath library function is called, the library is checked instead of in the program, which consumes extra gas.

Real Case 1: BearStaking	635 uint256[] public bears;
	……
	747 function getStakedBear(address wallet) public view returns(uint256[] memory){
	……
	750 for(uint256 i=0; i<bears.length; i++){
	……
	}

Real Case 2: Watchdog	949 function _transfer(address from, address to, uint256 amount) internal override { …… 953 require(balanceOf(to).add(amount) <= maxWalletLimit || isExcludedFromWalletLimit[to], "Transfer will exceed wallet limit");

Real Case 3: HGoldStaking 90Contract	157 for (uint i = 2; i < 2 + 2 * 20; i += 2) {
	……
	}

Real Case 4: PuffinPirates	527 contract PuffinPirates is ERC721Enumerable, Ownable { …… 536 bool public paused = false; 537 bool public revealed = false;
Real Case 5: TemplateCrowdsale	833 function _burn(address _who, uint256 _value) internal { 834 require(_value <= balances[_who]); …… 838 balances[_who] = balances[_who].sub(_value);

Fig. 13. Real cases.

6 Related Work

6.1 Pattern Analysis

There have been many studies on blockchain and SCs. Some studies have focused on the summary of gas-expensive patterns in SCs. Park et al. [8] focused on state variables and identified five gas-wasteful patterns related to storage operations. They proposed improved methods related to these patterns to identify and eliminate them. Zou et al. [12] learned developers' views on the current state of SCs and future challenges through the investigation. Qualitative and quantitative analysis showed that people should improve the operability of developing SCs to improve the developer experience. Marchesi et al. [31] summarized a series of design and development patterns that can save gas in SCs. All of the above works only studies on gas waste patterns and developers' understanding of SCs, and do not identify and solve gas-expensive patterns through practical tools.

6.2 Detecting Tools

There have multiple previous works aimed at improving gas waste.

Bytecode Level. Luu et al. [29] developed a symbol execution tool Oyente based on EVM bytecode, which is used to discover security vulnerabilities of SCs. Chen et al. [11] developed

GASPER based on Oyente, a tool for finding gas-costly patterns by detecting the EVM bytecode of SCs. And because GASPER is not comprehensive in discovering gas-costly patterns, Chen et al. [16] developed a tool based on symbolic execution, GasChecker, to find ten gas-inefficient patterns (three of which are EVM level). Chen et al. [17] developed GasReducer, which automatically detects 24 patterns at the bytecode level. Nagele et al. [18] optimized the EVM bytecode by encoding the operational semantics of the EVM instructions into SMT formulas to find cheaper bytecode instructions automatically. Albert et al. [19] developed GASOL, which judges the gas consumption of EVM instructions through two types of cost models. However, these existing works can only discover gas-expensive patterns from the bytecode level, which is not convenient for developers to modify these issues.

Source Code Level. Tikhomirov et al. [30] developed source code-based SmartCheck based on program analysis technology, which classifies and detects the code problems of security problems, function problems, and development problems of SCs. SmartCheck mainly aims at detecting SCs' vulnerabilities and only checks two gas waste problems. SmartCheck mainly aims at detecting vulnerabilities and only checks two gas waste problems. The first is useing byte[], and the second is the loop with a function call or an identifier inside the condition. Neuraturu et al. [32] developed a methodology that optimizes the loop constructs to decrease the gas consumption of SCs. This methodology combines loop summaries synthesis, syntactic transformations, and equivalence proofs. This work only considered optimizing gas waste patterns that can be applied to loops. Sorbo et al. [33] collected 19 code smells affected by the low efficiency of data storage and function implementation. They proposed GASMET, a tool for static evaluation of SCs code quality from the perspective of gas consumption. This work only focused on the wastage of gas on storage and the overall wastage on functionality, without specific analysis at the code level in terms of functionality.

This paper summarizes 12 gas-expensive patterns affected by storage, judgment, and loop at the source code level and develops GaSaver to detect them. GaSaver comprehensively explores most of the current gas-expensive patterns and shows developers what will produce gas waste via exploring all the code that could have the same results.

7 Conclusion

In this paper, we summarize 12 gas-expensive patterns. We detect each pattern and divide them into three levels according to the amount of gas waste degree. Then we develop `GaSaver` to detect gas-expensive patterns in smart contract source code. According to the experiment, 89.68% of 1172 SCs have gas-expensive patterns, while 94.27% of 1100 new SCs are affected, and 80.56% of 72 widely used SCs are affected. Moreover, we compare the new SCs with widely used SCs. The results show that the proportion of gas-expensive in widely used SCs is lower, indicating that the existence of gas-expensive in SCs is still common.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Bus. Rev.*, 2008, Art. no. 21260.

[2] Turing and A. Mathison, "On computable numbers, with an application to the entscheidungsproblem," *J. Math*, vol. 58, no. 345/363, 1936, Art. no. 5.

[3] Wikipedia, "Turing-completeness," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Turing_completeness

[4] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," vol. 3, no. 37, pp. 2–21, 2014.

[5] Ethereum.org, "Ethereum whitepaper," 2022. [Online]. Available: https://ethereum.org/en/whitepaper/

[6] D. Tapscott and A. Tapscott, *Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World*. Baltimore, MD, USA: Penguin, 2016.

[7] Etherscan, "Etherscan," 2020. [Online]. Available: https://etherscan.io/

[8] J. Park, D. Lee, and H. P. In, "Saving deployment costs of smart contracts by eliminating gas-wasteful patterns," *Int. J. Grid Distrib. Comput.*, vol. 10, no. 12, pp. 53–64, 2017.

[9] Ethereum.org, "Gas and fees," 2022. [Online]. Available: https://ethereum.org/en/developers/docs/gas/

[10] J. Frankenfield, "Gas (ethereum): How gas fees work on the ethereum blockchain," 2022. [Online]. Available: https://www.investopedia.com/terms/g/gas-ethereum.asp

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

[12] W. Zou et al., "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2019.

[13] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *Proc. IEEE 9th Int. Conf. Comput. Commun. Netw. Technol.*, 2018, pp. 1–4.

[14] K. Solorio, R. Kanna, and D. H. Hoover, *Hands-on Smart Contract Development with Solidity and Ethereum: From Fundamentals to Deployment*. Sebastopol, California: O'Reilly Media, 2019.

[15] P. Howson, "Tackling climate change with blockchain," *Nature Climate Change*, vol. 9, no. 9, pp. 644–645, 2019.

[16] T. Chen et al., "GasChecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1433–1448, Third Quarter 2020.

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

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

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

[20] R. Project, "Remix IDE," 2022. [Online]. Available: https://remix.ethereum.org/

[21] Wikipedia, "Blockchain," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Blockchain

[22] Coinbase, "What is ethereum," 2022. [Online]. Available: https://www.coinbase.com/price/ethereum

[23] Computerality, "Ethereum VM (EVM) opcodes and instruction reference," 2021. [Online]. Available: https://github.com/crytic/evm-opcodes

[24] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.

[25] Ethereum, "Solidity document," 2022. [Online]. Available: https://docs.soliditylang.org/en/develop/

[26] G. Wood, "Ethereum: A secure decentralized transaction ledger," Ethereum Project Yellow Paper, vol. 151, pp. 1–32, 2014.

[27] "Eth2 deposit contract," Accessed: May 24, 2022. [Online]. Available: https://etherscan.io/address/0x00000000219ab540356cbb839cbe05303d7705fa

[28] T. Parr, "Antlr," 2017. [Online]. Available: https://www.antlr.org/

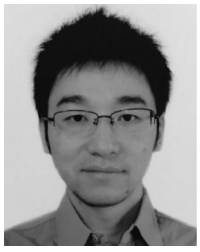
[29] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[30] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.

[31] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in ethereum," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng.*, 2020, pp. 9–15.
[32] K. Nelaturu, S. M. Beillahit, F. Long, and A. Veneris, "Smart contracts refinement for gas optimization," in *Proc. IEEE 3rd Conf. Blockchain Res. Appl. Innov. Netw. Serv.*, 2021, pp. 229–236.
[33] A. Di Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora, "Profiling gas consumption in solidity smart contracts," *J. Syst. Softw.*, vol. 186, 2022, Art. no. 111193.
[34] W. Hu, Z. Fan, and Y. Gao, "Research on smart contract optimization method on blockchain," *IT Professional*, vol. 21, no. 5, pp. 33–38, 2019.
[35] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022.
[36] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, 2018.
[37] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *Proc. IEEE Int. Conf. Blockchain*, 2020, pp. 281–290.
[38] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.
[39] S. B. LLC, "Bitcoin.com," 2022. [Online]. Available: https://markets.bitcoin.com/crypto/BTC/
[40] M. Gupta, "solidity gas optimization tips," 2018. [Online]. Available: https://medium.com/@MuditG/solidity-gas-optimization-tips-1658c2bf37e8
[41] W. Shahda, "Gas optimization in solidity part i: Variables," 2019. [Online]. Available: https://medium.com/coinmonks/gas-optimization-in-solidity-part-i-variables-9d5775e43dde
[42] M. Gupta, "Solidity tips and tricks to save gas and reduce bytecode size," 2019. [Online]. Available: https://blog.polymath.network/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6
[43] K. Zipfel, "How to write smart contracts that optimize gas spent on ethereum," 2020. [Online]. Available: https://betterprogramming.pub/how-to-write-smart-contracts-that-optimize-gas-spent-on-ethereum-30b5e9c5db85
[44] Ethereum.org, "Ethereum homestead documentation," 2016. [Online]. Available: http://www.ethdocs.org/en/latest/

Ziyi Zhao received the BEng degree in computer science and technology from Shanxi University, in 2016. He is currently working toward the MSc degree with the School of Software Engineering, Xi'an Jiaotong University, Xi'an, China. His current research interest is blockchain.

Jiliang Li received the Dr. rer. nat. degree in computer science from the University of Göettingen, Göettingen, Germany, in 2019. He is currently a researcher professor and PhD Supervisor with the School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China. His research interests include information security, cryptography, blockchain, and IoT security.



Zhou Su has published technical papers, including top journals and top conferences, such as *IEEE Journal on selected areas in communications*, *IEEE transactions on information forensics and security*, *IEEE transactions on dependable and secure computing*, *IEEE transactions on mobile computing*, *IEEE/ACM transactions on networking, and infocom*. His research interests include multimedia communication, wireless communication, and network traffic. He received the best paper award of International Conference IEEE ICC2020, IEEE BigdataSE2019, and IEEE CyberSciTech2017. He is an associate editor of *IEEE internet of things journal*, *IEEE open journal of com~puter society*, and *IET communications*.

Yuyi Wang received the PhD degree in computer science from the Katholieke Universiteit Leuven, Leuven, Belgium, in 2015. He is currently a researcher with the ETH Zürich, Zürich, Switzerland, and also a youth scientist with the CRRC Zhuzhou Institute, Hunan, China. His research interests include algorithm theory, machine learning, and blockchain.